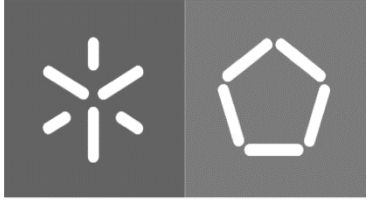
Universidade do Minho
Escola de Engenharia

Hugo Miguel Eira Araújo

ILTZVisor: A Lightweight TrustZone-assisted Hypervisor for low-end ARM Devices

Outubro de 2018



Universidade do Minho
Escola de Engenharia

Hugo Miguel Eira Araújo

ILTZVisor: A Lightweight TrustZone-assisted Hypervisor for low-end ARM devices

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Doutor Sandro Pinto

Outubro de 2018

Declaração do Autor

Nome: Hugo Miguel Eira Araújo
Correio Eletrónico: a72139@alunos.uminho.pt
Cartão de Cidadão: 14887613

Titulo da dissertação: lLTZVisor: A Lightweight TrustZone-assisted Hypervisor for low-end ARM Devices
Ano de conclusão: 2018
Orientador: Doutor Sandro Pinto

Designação do Mestrado: Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia Eletrónica Industrial e Computadores
Área de Especialização: Sistemas Embebidos e Computadores
Escola de Engenharia
Departamento de Eletrónica Industrial

Universidade do Minho, 29/10/2018

Assinatura: Hugo Miguel Eira Araújo

Acknowledgements

The accomplishment of this master's thesis involved great contributions from several people, whom I will be eternally grateful.

Foremost, I would like to express my gratitude to my advisor Dr. Sandro Pinto for giving me the opportunity to develop this groundbreaking work and for all the guidance and continuous support throughout this great journey. Thank you for all the confidence placed on me, and especially, for all the motivation and advices given during difficult times.

I would like to also thank Dr. Adriano Tavares, my professor, whom I appreciate all the given advices and reviews. And of course, thanks for all the music tips that helped me to increase productivity. To Daniel Oliveira and José Martins a special thanks for helping me and for all the recommendations and reviews.

To all my fellow lab-mates in the ESRG group, a massive thank you. Andersen Bond, Ângelo Ribeiro, Francisco Petrucci, José Silva, Nuno Silva, Pedro Machado, Pedro Ribeiro, Ricardo Roriz and Sérgio Pereira, it was a pleasure working alongside you. Thanks for all the funny moments we shared, for all the stimulating discussions we have had and for the good team spirit provided in the lab. And of course, thanks again to Sérgio Pereira and Andersen Bond - "Deutschgang" - whom I shared unforgettable moments during our semester abroad in Germany.

To every friend and family that were supportive and encouraging with me throughout this journey, my sincere thank you.

Last but not least, I want to leave a special and huge thank you to my fathers and brothers for the unconditional love and constant support throughout my life.

Abstract

Virtualization is a well-established technology in the server and desktop space and has recently been spreading across different embedded industries. Facing multiple challenges derived by the advent of the Internet of Things (IoT) era, these industries are driven by an upgrowing interest in consolidating and isolating multiple environments with mixed-criticality features, to address the complex IoT application landscape. Even though this is true for majority mid- to high-end embedded applications, low-end systems still present little to no solutions proposed so far.

TrustZone technology, designed by ARM to improve security on its processors, was adopted really well in the embedded market. As such, the research community became active in exploring other TrustZone's capacities for isolation, like an alternative form of system virtualization. The lightweight TrustZone-assisted hypervisor (LTZVisor), that mainly targets the consolidation of mixed-criticality systems on the same hardware platform, is one design example that takes advantage of TrustZone technology for ARM application processors. With the recent introduction of this technology to the new generation of ARM microcontrollers, an opportunity to expand this breakthrough form of virtualization to low-end devices arose.

This work proposes the development of the lLTZVisor hypervisor, a refactored LTZVisor version that aims to provide strong isolation on resource-constrained devices, while achieving a low-memory footprint, determinism and high efficiency. The key for this is to implement a minimal, reliable, secure and predictable virtualization layer, supported by the TrustZone technology present on the newest generation of ARM microcontrollers (Cortex-M23/33).

Resumo

Virtualização é uma tecnologia já bem estabelecida no âmbito de servidores e computadores pessoais que recentemente tem vindo a espalhar-se através de várias indústrias de sistemas embebidos. Face aos desafios provenientes do surgimento da era Internet of Things (IoT), estas indústrias são guiadas pelo crescimento do interesse em consolidar e isolar múltiplos sistemas com diferentes níveis de criticidade, para atender ao atual e complexo cenário aplicativo IoT. Apesar de isto se aplicar à maioria de aplicações embebidas de média e alta gama, sistemas de baixa gama apresentam-se ainda com poucas soluções propostas.

A tecnologia TrustZone, desenvolvida pela ARM de forma a melhorar a segurança nos seus processadores, foi adoptada muito bem pelo mercado dos sistemas embebidos. Como tal, a comunidade científica começou a explorar outras aplicações da tecnologia TrustZone para isolamento, como uma forma alternativa de virtualização de sistemas. O "lightweight TrustZone-assisted hypervisor (LTZVisor)", que tem sobretudo como fim a consolidação de sistemas de criticidade mista na mesma plataforma de hardware, é um exemplo que tira vantagem da tecnologia TrustZone para os processadores ARM de alta gama. Com a recente introdução desta tecnologia para a nova geração de microcontroladores ARM, surgiu uma oportunidade para expandir esta forma inovadora de virtualização para dispositivos de baixa gama.

Este trabalho propõe o desenvolvimento do hipervisor lLTZVisor, uma versão reestruturada do LTZVisor que visa em proporcionar um forte isolamento em dispositivos com recursos restritos, simultâneamente atingindo um baixo *footprint* de memória, determinismo e alta eficiência. A chave para isto está na implementação de uma camada de virtualização mínima, fiável, segura e previsível, potencializada pela tecnologia TrustZone presente na mais recente geração de microcontroladores ARM (Cortex-M23/33).

Contents

List of Figures

List of Algorithms

Acronyms

AAPCS	Procedure Call Standard for the ARM Architecture
ABI	Application Binary Interface
ACTLR	Auxiliary Control Register
ADC	Analog-Digital Converter
AIRCR	Application Interrupt and Reset Control Register
AMP	Asymmetric Multiprocessing
API	Application Programming Interface
CPU	Central Processing Unit
DAC	Digital-Analog Converter
DoS	Denial of Service
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
ES	Embedded System
ESRG	Embedded Systems Research Group
FIFO	First-In First-Out
FIQ	Fast Interrupt Request
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit
FVP	Fixed Virtual Platform
GIC	Generic Interrupt Controller
GPOS	General Purpose Operating System
HMI	Human-Machine Interface
IDAU	Implementation Defined Attribution Unit
IoT	Internet of Things
IP	Intellectual Property
IPO	Initial Public Offering
IRQ	Interrupt Request
ISA	Instruction Set Architecture

ISR	Interrupt Service Routine
IVI	In-Vehicle Infotainment
LLC	Last-Level Caches
LR	Link Register
MCU	Microcontroller Unit
MMU	Memory Management Unit
MPS2+	Microcontroller Prototyping System 2
MPSoC	Multiprocessor System-on-Chip
MPU	Memory Protection Unit
MSP	Main Stack Pointer
NSC	Non-Secure Callable
NVIC	Nested Vectored Interrupt Controller
OS	Operating System
PC	Program Counter
PTE	Page Table Entry
QSPI	Quad Serial Peripheral Interface
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTOS	Real-Time Operating System
SAU	Secure Attribution Unit
SCB	System Control Block
SCTLR	System Control Register
SG	Secure Gateway
SGI	Software Generated Interrupt
SMC	Secure Monitor Call
SMP	Symmetric Multiprocessing
SoC	System-on-Chip
SP	Stack Pointer
SPSR	Saved Program Status Register
SPUMONE	Software Processing Unit, Multiplexing ONE into two
SRAM	Static Random Access Memory
SVC	Supervisor Call
TCB	Trusted Computing Base
TCM	Tightly-Coupled Memory
TLB	Translation Lookaside Buffer
TTC	Triple Timer Counter

TZASC	TrustZone Address Space Controller
TZMA	TrustZone Memory Adapter
TZPC	TrustZone Protection Controller
vCPU	Virtual Central Processing Unit
VLSI	Very Large Scale Integration
VM	Virtual Machine
VMCB	Virtual Machine Control Block
VMM	Virtual Machine Monitor
VTOR	Vector Table Offset Register
WCET	Worst-Case Execution Time
WSN	Wireless Sensor Network

1. Introduction

The embedded domain started recently following the computing industry, in which system virtualization is nowadays a mainstream tool, as indicated by billion-dollar Initial Public Offerings (IPOs) and sales of startup companies for hundreds of millions [Hei08]. Despite that some years ago adopting virtualization techniques in Embedded Systems (ESs) seemed to be both a distant and unnecessary reality [AH10], the rise of computational power of these ESs and consequentially, the demand for newer and modern functionalities, proved wrong. It is then assumed that modern ESs are increasingly taking on characteristics of general-purpose systems [Hei08]. For example, mobile phones and consumer electronics products are becoming very sophisticated, in contrast to former days, where most ESs just had simple functionalities and were providing only a few services [KYK+08].

Since new services are being added to ESs and the functionalities are becoming rich year after year [KYK+08], while simultaneously its usage in our daily lives is seriously increasing, attackers will try to take advantage of any weak link in their security [FA16] in order to compromise the system. A security breach thus can result in physical side effects, including property damage, personal injury, and even death, due to the direct interaction of an embedded device with the physical world. Backing out financial transactions can repair some enterprise security breaches, but reversing a car crash is not possible [Koo04, FMR11]. This is where security shows as a new and emerging area of research on the ESs field, in which not too long ago its devices were considered secure and out of reach of attacks, as opposed to general purpose digital systems which were prone to various types of attacks [FA16]. The situation then turned upside down after several incidents related to ESs were reported, and consequently ESs security gained importance in 1990s. Since then, embedded devices could be subjected to remote attacks, a result of the emergence of networked ESs and the source of most security breaches [FA16, PW08]. In addition, many of the inherent characteristics of ESs have direct impact on security-related issues [PW08]. These characteristics are directly tied

to the following vulnerabilities:

- **Limited Resources** - These limitations provide an avenue of attack that can easily cause a Denial of Service (DoS) attack by consuming any of these resources, e.g. with the intuition of seeking to drain the battery (if device is battery powered) to then cause system failure [FA16, Koo04, FMR11, PW08].
- **Timing Deadlines** - By attacking an embedded device aiming to disrupt system timing, just a matter of a fraction of seconds could cause a deadline miss and a loss of control loop stability, thus resulting on property or even life loss [FA16, Koo04].

This trend attracted the industry's and academia's attention in recent years to develop and present a new variety of different solutions to address these security problems [PW08].

Succeeding the technological advance, multi-core devices started being developed, justified by the requirements of low-power and multi-tasking from systems that make use of new and demanding functionalities like networking, communications, signal processing, multimedia and others. These type of devices emerged in the past two decades, and are technically called Multiprocessor System-on-Chips (MPSoCs). They use multiple Central Processing Units (CPUs) along with other hardware subsystems, which is basically a Very Large Scale Integration (VLSI) system that incorporates most or all the components necessary for an application, a unique branch of evolution in computer architecture [WJM08]. This multi-processed platforms trend is being increasingly adopted in ESs and becoming each more a viable choice to implement. Embedded developers are for this reason forced to change the way they design their systems, since the use of multi-core processors and virtualization techniques bring several key advantages [AH10].

Virtualization is a technique that was first adopted on general-purpose systems, and only recently became popular in the ESs space. Traditionally, ESs were characterized by simple functionality, a single purpose, no or very simple user interface, and no or very simple communication channels. Although, modern ESs feature a wealth of functionality. Being that advanced and offering various relevant features, it is understandable the attraction of virtualization in the ESs context [Hei08], since it is a great solution that allows the consolidation of different workloads on the same platform, fitting practically and efficiently on recent sophisticated embedded devices. For instance, applications of the Internet of Things (IoT)

industry, which typically communicate over closed industrial networks, are connected to the Internet and contain code from multiple developers. Thus, effort not only must be done in order to guarantee the classic hard timing requirements of critical control tasks, but also to ensure the integrity of the multiple tasks related to maintenance and configuration updates. Problems that motivate the creation of execution environments such as the IIoTEED [PGP+17], a way of guaranteeing reliability, security and predictability to modern smart sensor nodes, which contain environments of mixed-criticality.

1.1 Problem Statement

Nowadays, virtualization in ESs presents as a solution for the next-generation embedded devices, allowing concurrent existence and operation of multiple Operating Systems (OSs) on the same hardware platform, despite its several differences and limitations implied by embedded devices, in comparison to general-purpose systems and enterprise servers [Hei08]. Also, software complexity dramatically increased due to growing functionality of the embedded devices. That is why it is very common to run general-purpose applications on ESs as well as to use applications written by developers that have little or no acknowledge at all about the ESs constraints, creating a demand for high-level application-oriented operating systems with commodity Application Programming Interfaces (APIs) [AH10, Hei08]. Nevertheless, embedded devices are still real-time systems and have concerns on energy consumption, memory usage and factoring cost. At the same time, they are increasingly used in mission- and life-critical scenarios due to their ubiquity, in a degree that is becoming hard to imagine living without them. So, it is possible to say that virtualization could bring advantages by increasing modern embedded devices' safety, reliability and security, which represent the highest and growing modern ESs' requirements [AH10, Hei08].

Furthermore, virtualization technology stood as a major assist to address recent demands on embedded devices, by providing a solid infrastructure to combine uneven real-time characteristics with general-purpose applications, allowing the co-existence and execution of a Real-Time Operating System (RTOS) and General Purpose Operating System (GPOS) on the same hardware platform. Not only this addresses the conflicting requirements of high-level APIs for application programming and real-time constraints, but it is also a way to save engineering cost and development time, since GPOSes already provide highly abstracted APIs

and software resources such as middleware, network applications and development tools [KYK+08, Hei08, Hei07]. Therefore, if the RTOS is able to meet its timing deadlines, providing real-time functionality to the device, the application OS can provide the required commodity APIs and high-level functionality suitable for application programming [Hei08]. Simultaneously, this ability to run concurrent OSes on a single processor core can reduce the bill of materials, especially for lower-end devices [Hei07].

Security presents as another strong motivation towards virtualization adoption. As said previously, the likelihood of an application OS being compromised by an attacker increased dramatically. By recurring to virtualization technology, damage can be minimized by running such an OS in a Virtual Machine (VM) which limits access to the rest of the system [Hei08]. This is done by providing strong isolation between guest OSes (or VMs), running under a Virtual Machine Monitor (VMM) (or hypervisor), assuring the availability of the device even when a guest OS fails, since any compromised OS cannot be propagated and interfere with other OS domains. User attacks will only be able to cause damages at the user OS, thus keeping the RTOS and specific components safe [HSH+08, Hei07, AH10]. With virtualization, the high-level OS is de-privileged and unable to interfere with data belonging to other subsytems, because each subsystem is encapsulated into a VM. Hence, virtualization can be used to enhance security [Hei07]. Moreover, ARM TrustZone [ARM09], an hardware security extension introduced to ARM processors, is a technology centered around the concept of separating the system execution into the secure and non-secure worlds. Being ARM processors highly adopted in the embedded market, TrustZone became a very hot topic because allowed manufacturers to improve the security of their products. As such, research community started being active in exploring ways to leverage TrustZone for isolation, including the use of TrustZone to implement an alternative form of system virtualization [SHT10, PPG+17c] which combines both the topics discussed.

As multi-core chips proliferate, as the incremental cost of a core is dropping dramatically and as two lower-performance cores are likely to have lower average power consumption than a single higher-perfomance one, virtualization techniques, which can be easily migrated from a single-core architecture, will suit even better a platform incorporating this multi-core architecture [Hei08]. Efficiently managing power consumption and workload balancing are emphasizing characteristics that can be explored by multiprocessing configurations like Asymmetric Multiprocessing (AMP) and Symmetric Multiprocessing (SMP). In a real hypervisor application, the former configuration would dictate that each core had its own

separate OS and thus the responsibility for scheduling its own tasks. On the other hand, the latter configuration could mean that a guest OS was allowed to be mapped onto multiple cores, enabling load balancing across the processors [AH10]. This is a great example where a hypervisor could dynamically add cores to an application domain which requires extra processor power, or could manage power consumption by removing processors from domains and shutting down idle cores [Hei08].

Summing up, virtualization presents as an essential technology to the actual embedded enterprise world, mainly by granting reduction of costs due to its encourageable to use inherent advantages [AH10].

1.1.1 Goals

Several virtualization solutions have been studied to address recent demands of modern embedded devices to consolidate on the same platform uneven heterogeneous environments with different requirements. Developing a virtualization layer in order to combine mixed-criticality environments is the approach used by some hypervisors [PPG+17c, KYK+08, HSH+08, SHT10, KLJ+13]. However, these existing solutions focus on high-end platforms and depend on hardware features which typically are not available on Microcontroller Units (MCUs).

The in-house developed lightweight TrustZone-assisted hypervisor [PPG+17c] (LTZVisor) is one of the solutions stated previously. This open-source hypervisor exploits the ARM TrustZone security extensions [ARM09] to implement an infrastructure that is capable of consolidating mixed-criticality systems. However, low-end devices are not supported. The main goal of this thesis is to develop an hypervisor for the new-generation of low-end ARM devices (Cortex-M23/33), which contain a similar version of the TrustZone security extension exploited by LTZVisor. Therefore, this thesis aims to benefit from LTZVisor's design ideas by creating a completely re-factored version of the LTZVisor. A virtualization layer that borrows the LTZVisor's design principles but that targets instead the modern low-end ARM processors family, the low-end lightweight TrustZone-assisted hypervisor (lLTZVisor).

From a high-level perspective, modern ARM MCUs contain a similar version of the TrustZone security extensions present on other higher-end ARM processors. However, the two versions present different underlying mechanisms, which makes it impossible to directly apply the same type of assisted virtualization to every

ARM MCU processor. This means that the lLTZVisor hypervisor will face several key challenges in order to shift to a TrustZone-assisted virtualization version for MCUs, along with coping with real-time determinism and the typical size and timing characteristic constraints of the low-end devices.

lLTZVisor aims to bring virtualization for the newer ARM microcontroller platforms with single or multi-core processors, and to prove that these devices can also retrieve advantages as [AH10]:

- Allow two OSes (RTOS + application OS) to run on a single platform;
- Improve security by isolating the VMs, and;
- Increase system flexibility.

Finally, this thesis has the intent to prove whether or not virtualization could be adopted in the enterprise market of microcontrollers and IoT devices. And, most importantly, to start a discussion about the exploitation of TrustZone technology on low-end ARM devices aiming virtualization, presenting the drawbacks of this approach, along with its advantages.

1.2 Document Structure

The structure of this remaining document is as described next. Second chapter is composed by an analysis overview of concepts and ideas of virtualization technology, along with the study of some available hypervisor solutions on the market. Still part of this chapter, is the description of the target CPU's architecture, the technology (ARM TrustZone security extensions) supporting the hypervisor (lLTZVisor) and a description of the LTZVisor, from which it was borrowed the virtualization technique and its core principles that are fundamental for the work developed in this thesis. Third chapter presents the research tools and platforms used on the development of the lLTZVisor hypervisor. Fourth chapter provides the overview, design and implementation of the lLTZVisor, with details that relate from the initial stage design of the project to every implemented detail. Fifth chapter presents the extensive evaluations performed to the developed hypervisor, accompanied by the results that demonstrate the overall performance that an application running above the lLTZVisor can reach, along with a deep study of possible impacts on predictability and determinism. Ending the document, the

Sixth chapter resumes the developed work of this thesis, and concludes discussing both future potential improvements to the hypervisor.

2. State of the Art

In this chapter, the theoretical virtualization principles are presented, following with an overview focused on available embedded hypervisors solutions. Additionally, the recent ARM CPU architecture for microcontrollers and ARM TrustZone technology is laid out. The subsequent order of topics intends to first introduce the reader to the groundwork concepts and only then to focus on the recent ARM technological advances, that potentially allow the practical use of those concepts.

2.1 Virtualization

Virtualization was first introduced in the 1960s by researchers at IBM, where they made the first ever Virtual Machine (VM), an efficient and isolated duplicate of a real machine [Hei07], to enable repeated interface access to a mainframe computer. Since then, virtualization has been a well-covered research topic [Sch14, OS15]. Nowadays, this technology is a mainstream tool in the server and desktop space, presenting huge benefits in terms of load balancing, power managing and service consolidation. In ESs space, virtualization has also the potential to be a game-changer, thus it is likely to become more widespread in the next few years [PPG+17a, Hei11].

> *"Virtualization is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others."* [Ami04]

Despite the concept of virtualization being very broad, simply put, virtualization is a technique that separates a resource or request for a service from the underlying physical delivery of that service [VMW06, IBM07]. It is a technique that targets the efficient use of the available computing resources. With the use of it, resources

can be consumed more effectively than conventional bare-metal setups, which use physical machines for isolating different parts of application infrastructure [Sch14]. The main advantage is that, if a VM fails, the other ones are kept safe at a reasonable cost [AH10].

Among many ways in which virtualization can be used and implemented, two of them really stand out. These two leading software virtualization approaches to date have been full virtualization and paravirtualization [VMW07, OS15, Sch14]. Both of which differ in terms of trade-offs between flexibility and performance, respectively incurring a higher performance cost and a higher design cost [PPG+17a].

Full Virtualization
This approach allows VMs to run unmodified, not requiring a single kernel adjustment. It maintains the existing investments in operating systems and applications, together with providing a non-disruptive migration to virtualization environments [Sch14, VMW07]. A complete look-alike of the real hardware is virtualized to embrace this no VM modifications policy [OS15].

Para-virtualization
Although requiring special modifications to the VMs' kernels, as opposing to the previous approach, this approach targets to optimize for performance in the virtual environment. It eliminates the need for binary translation of a VM and offers potential performance advantages for certain workloads. Hardware limitations and efficiency reasons led to the widespread use of para-virtualization [Sch14, VMW07, HL10]. One potential downside of this approach is that such modified VMs cannot ever be migrated back to run on physical hardware [VMW06]. This approach organizes the access to hardware resources in aid of the VM, rather than emulating the entire hardware environment in the software [OS15].

2.1.1 Hypervisor

The hypervisor, also known as the Virtual Machine Monitor (VMM), is the basic software, firmware or hardware which implements virtualization. It provides the means to run multiple OSes on a single physical machine, each isolated from one another [AH10], by creating an environment where resources are provided virtually, either by temporarily mapping them to physical resources, or emulating them

[HL10]. Its main purpose is to monitor the VMs (which can also be called as domains or guests) running on top of the hypervisor, see Figure 2.1. Fundamentally, this enables more than one guest machine to utilize the hardware resources of a single host machine [Sch14, OS15].

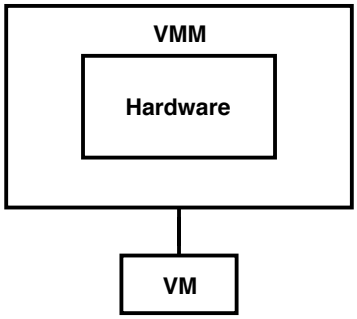


Figure 2.1: The Virtual Machine Monitor. Adapted from [PG74].

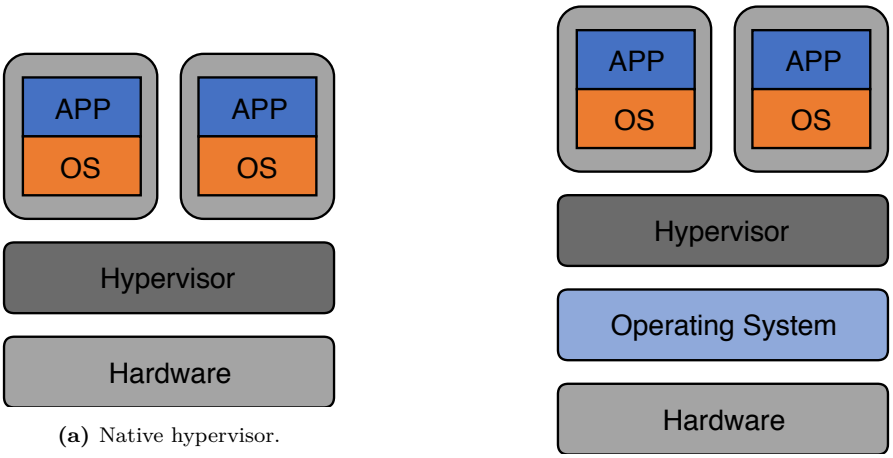
In order to contribute to making virtualization highly useful in practice, an hypervisor needs to ideally have three fundamental characteristics [Hei07, PG74]:

1. **Compatibility** - Provide the VMs an identical environment as to the original machine, so software that runs on the real machine will run on the hypervisor and vice-versa;
2. **Reliability** - Guarantee minor decreases in speed to the VMs running on the hypervisor, achieving practicability from a performance point of view, and;
3. **Security** - Ensure that software cannot break out of the VM by giving total control of the systems resources do the hypervisor.

Furthermore, para-virtualization supporting hypervisors have to provide extra APIs, called hypercalls, which generally are more high-level than the hardware Application Binary Interface (ABI), in order to fit the guest OSes ports [HL10].

Generally, an hypervisor can be classified as either one of two distinct types, namely: Type 1 - Native or Bare-metal (Figure 2.2a), and; Type 2 - Hosted (Figure 2.2b) [OS15, AH10].

Native hypervisors (see Figure 2.2a) run directly above the hardware of the host machine, hence they are also known as hardware level virtualization or bare-metal. In this scope, the hypervisor itself can be considered as an OS, since this piece of software is supposed to run on the highest privilege mode and will be in charge of managing multiple VMs, just like an OS manages different tasks [AH10, OS15]. This approach is used mostly by embedded devices.



(a) Native hypervisor.

(b) Hosted hypervisor.

Figure 2.2: The two hypervisor types. Adapted from [Web].

On the other hand, hosted hypervisors (see Figure 2.2b) are installed on already existing OSes and virtualize other OSes that are above them. Thus, it gets commonly called as an Operating System level virtualization. All the VMs are abstracted from the host OS. As this hypervisor architecture heavily relies on the host OS for device support and physical resource management, any problem occurring with the host will affect the VMs and the hypervisor itself [AH10, OS15].

In both cases the VMs must behave exactly the same as if they were running on real hardware. Then, the creation of each scenario is up to the hypervisor and the real hardware of the machine, which will be responsible for dealing with instruction coming from the VM [AH10].

2.1.2 Embedded Hypervisors

Within the scope of this thesis, it is vital to understand which are the fundamental approaches applied to embedded devices' dedicated hypervisors, so that the lLTZVisor hypervisor is able to follow certain design characteristics that were already studied and discussed.

This variation on the paradigm (of hypervisors), caused by the ubiquity of embedded devices and their recent demands [Hei08], brought a change in direction, that focus more on real-time capabilities, Worst-Case Execution Time (WCET) scenarios and OS heterogeneity [AH10]. Unlike former solutions as seen on enterprise servers or general-purpose computing.

After a broad research on the main solutions developed throughout the years, the following sections will contain some hypervisors that in a certain way or another, helped on the development of the lLTZVisor hypervisor. Either by containing similar principles, comparable target architectures and for the most part, by the main goals.

2.1.2.1 SPUMONE

The Software Processing Unit, Multiplexing ONE into two (SPUMONE) [KYK+08] is by nomenclature a "Lightweight CPU Virtualization Layer for Embedded Systems". Its approach is to build an hybrid OS by using existing virtualization techniques, intending to run a RTOS and a GPOS on a hardware virtualization layer, with the goal of providing a system capable of using both real-time subsystems and multimedia subsystems. Essentially, it was developed assuming that an hybrid OS environment (an approach that tries to take advantage of the combination of two OSes with different characteristics) is a solution to the increased growth of engineering cost of ESs' software associated to recent demands of services added to embedded devices, by providing an infrastructure that enables the re-usability of different OSes' software resources [KYK+08].

SPUMONE strictly follows guidelines that meet the necessary requirements for an hybrid OS platform [KYK+08]:

1. **Minimal guest OS code modification** - This requirement follows the idea that as few code modifications to guest OSes are made, the most easily will be to go along with up-to-date kernel versions, hence reducing engineering costs.

2. **Lightweight virtualization layer** - The virtualization layer must be kept lightweight, in order to minimize the performance degradation and prevent the increase of the worst case response time of the guest OSes. Additionally, software is prone to bugs as it grows large, and by keeping the virtualization layer lightweight, reliability increases.

3. **Independently reboot OSes** - This concerns system availability. By isolating each guest OS independently, one would be able to provide service continuously even if the other one encountered a fatal error, which would result on its reboot but without jeopardizing the other guest OSes. In that case, the OS that encountered a fatal error should be able to reboot independently and the availability of the whole system would not be compromised.

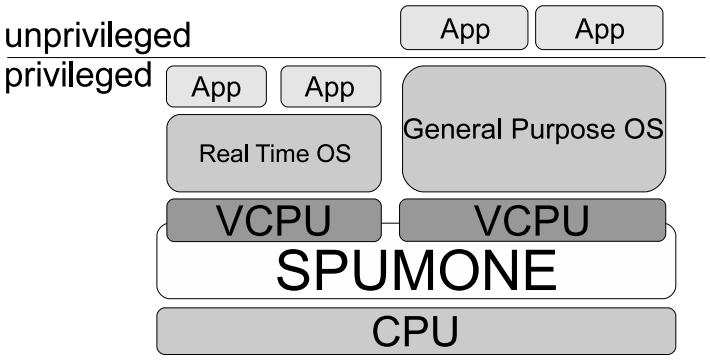


Figure 2.3: The overview of SPUMONE. Reproduced from [KYK+08].

SPUMONE design approach is to virtualize a single real CPU so it provides multiple Virtual Central Processing Unit (vCPU) to its guests, as shown in Figure 2.3. Moreover, SPUMONE assigns every hardware device (except the CPU) to each of the guest OSes instead of sharing or virtualizing them, in order to comply with the guidelines presented earlier. This is done in order to minimize the virtualization layer and to prevent the degradation of the worst case response time of the guest RTOS. To minimize the engineering cost due to modifications of the source code of guest OSes, SPUMONE minimizes the emulation overhead for privileged instructions by letting the guest OSes run in privilege mode and execute most privileged instructions. This decreases performance degradation minimally, and eliminates the need to replace almost all of the privileged instructions in the guest OS by the virtualization layer APIs. Lastly, SPUMONE supports a feature to reboot guest OSes independently from other parts of the system, respecting in full the requirement 3 [KYK+08].

2.1.2.2 Xen on ARM

Xen on ARM [Xen18] was born after the proposal to design a system virtualization layer for the ARM CPU architecture. It was implemented by porting one of the most popular open source solutions, the Xen Hypervisor [BDF+03], which initially only supported x86 architecture.

As efficiency is considered a major concern in embedded virtualization, para-virtualization was the chosen approach by the team behind Xen on ARM [Xen18]. This means that Binary translation, which is generally used in full virtualization approaches, is consequently excluded from the design because of its inherent resource expensivenesses that a restricted device cannot afford to have. Xen Hypervisor [BDF+03], a publicly available para-virtualization solution, was chosen

for this port because of its rather simple interface and, mostly, due to its architecture independent common components of VMM such as hypercall interface, VM scheduling, VM controls, and memory management [HSH+08].

Since initially the Xen Hypervisor [BDF+03] was developed for a x86 CPU architecture, ARM CPUs are not able to get a direct port because architectures like x86 feature rich functions designed for desktops and servers, while in comparison, ARM's architecture lacks functionality, mostly regarding virtualization capabilities. Nonetheless, the Xen on ARM team was still able to circumvent these issues and finish the port to ARM architecture. On recent architectures, ARM improved the virtualization capabilities of its CPUs, which enabled a newer architecture for Xen on ARM [Xen18]. This virtualization solution became much cleaner, simpler and fitted the hardware better, unlike initial designs where some additional logic was required [Xen18].

Fundamentally, Xen on ARM [Xen18] is a lightweight, high-performance and type-1 open-source hypervisor for ARM processors with the ARMv7-A architecture. It virtualizes CPU, memory, interrupts and timers, providing each VM with one or more vCPUs, a fraction of the memory of the system, a virtual interrupt controller and a virtual timer. This hypervisor allows users to adjust and create several different configurations on the system by privileging certain VMs and assign device's accesses. For example, it allows users to run a RTOS alongside the main OS to drive a device that has real-time constraints or even allow to separate and isolate critical functionalities from less critical ones. Also, although it uses a para-virtualization approach, no changes are required to the OS kernel of the VMs, but only a few new drivers are needed to get the paravirtualized frontends running and to obtain access to devices [HSH+08, Xen18].

2.1.2.3 OKL4 Microvisor

One example of a microkernel-based embedded hypervisor (called microvisor) solution is the OKL4 Microvisor [HL10], an open-source system software platform developed by the Open Kernel Labs company [Gen]. This hypervisor is of Type-1 and runs on single- and multi-core platforms based on ARM, x86 and MIPS processors. The idea behind combining microkernels and hypervisors, which are both designed at low-level foundations for larger systems and present different objectives, is to enable building a type of kernel that satisfies both technology's objectives. Hence a kernel such as OKL4 is defined as a microvisor [HL10].

The microvisor model goes along with the goal of supporting virtualization with the lowest possible overhead, so, its abstractions are designed to model hardware as closely as possible, for example [HL10]:

- **vCPU** - The microvisor's execution abstraction is similar to a VM with one or more vCPUs, on which the guest OS can schedule activities;
- **Memory Management Unit (MMU)** - The memory abstraction consists of a virtual MMU, which the guest OS uses to map virtual to (guest) physical memory;
- **I/O** - The I/O abstraction consists of memory-mapped virtual device registers and virtual interrupts, and;
- **Communication** - Virtual interrupts (for synchronization) and channels, functioning as bi-directional FIFOs buffers allocated in user space which are used to abstract the interaction between guests.

The OKL4 Microvisor [HL10] is designed to support a mixture of real-time and non-real-time software running on top. It is also designed to support para-virtualization, since most cores for embedded use do not support full virtualization. Latest trends suggest that hypervisors are becoming more microkernel-like. The tendency to move drivers into user-space, as a way to reuse legacy drivers, is one indication. This is one of the reasons why the team behind OKL4 believes in the combination of microkernels and hypervisors, because both have concepts that share growing similarities. OKL4 Microvisor is therefore the convergence point between those two models that meets the goals of minimal overhead for virtualization (hypervisor) and minimal size (microkernel) [HL10].

2.1.2.4 TrustZone-assisted Hypervisors

The idea of using TrustZone technology to implement hardware-assisted virtualization solutions for (real-time) embedded systems applications is not new. Frenzel et al. [FLWH10] pioneered research in this domain by proposing the use of TrustZone for implementing the Nizza secure architecture [HHF+05]. SafeG [SHT10], SASP [KLJ+13], and LTZVisor [PPG+17c] (detailedly presented on Section 2.2) are dual-OS solutions which take advantage of TrustZone extensions for virtualization.

SafeG [SHT10] is an open-source solution which allows the consolidation of two different execution environments: an RTOS such as TOPPERS ASP kernel, and

a GPOS such as Linux or Android. SASP [KLJ+13] implements a lightweight virtualization approach which explores TrustZone technology to provide isolation between a control system and an In-Vehicle Infotainment (IVI) system. LTZVisor [PPG+17c] is an open-source lightweight TrustZone-assisted hypervisor mainly targeting the consolidation of mixed-criticality systems. While the lack of scalability was the main reason that led several researchers to perceive TrustZone as an ill-guided virtualization technique for many years, RTZVisor [PPG+17b] and μRTZVisor [MAC+17] have recently demonstrated how multiple OS instances are able to coexist, completely isolated from each other, on TrustZone-enabled platforms. All aforementioned works are implemented in Cortex-A processors and targeting mid- to high-end applications.

2.2 LTZVisor Overview

In the present section, the hypervisor which serves as the basis for this thesis to rely on is presented, be it regarding its design ideas, principles and architectural structure, or mainly by the fact of standing out as an hypervisor solution assisted by the ARM TrustZone technology, just as what this thesis strives for. The LTZVisor [PPG+17c] is the hypervisor at stake, thus, the following sections will contain information regarding this hypervisor's ideas and principles, completed by a final summary to highlight the major key points that will be borrowed to develop the aimed TrustZone-enabled lightweight hypervisor.

The LTZVisor is an hypervisor solution that carries an overall goal of studying, evaluating and understanding the usefulness of the exploitation of the ARM TrustZone technology to assist the development of a virtualization layer, pointing out both the benefits and limitations imposed by the technology used on this context. Despite the idea not being completely new, this solution still bets on leaving a legacy behind in order to influence future hypervisor solutions that the ARM TrustZone technology is appropriate to assist implementations of virtualization layers on embedded devices.

Due to the physical core virtualization into two virtual cores, proportioned by the TrustZone technology, two separate execution domains can be achieved. These domains, namely secure and non-secure world, can translate into acting as VMs running above the underlying virtualization layer. This technology allows the execution to switch between worlds during run-time, and every time they occur, the processor jumps to monitor mode, an extra processor mode added by TrustZone,

that will serve as a bridge between the VMs. During this mode, the processor state is always considered secure, thus ideal to preserve the processor state during a world switch. In order to enter monitor mode, either the instruction Secure Monitor Call (SMC) must be executed or an exception of the secure world is configured to be handled on monitor mode.

Additionally, on TrustZone supported processors, some special registers are banked between the two security processor states, ensuring a strong isolation between them. Also, other critical processor registers are only visible to the secure world, while others require access permissions under control of the secure world. Regarding security on the memory infrastructure, TrustZone is able to extend it there by introducing TrustZone Address Space Controller (TZASC) and TrustZone Memory Adapter (TZMA). They respectively enable Dynamic Random Access Memory (DRAM) partitioning into different specific memory regions as secure or non-secure, and similar functionality for off-chip memory devices such as Read Only Memory (ROM) or Static Random Access Memory (SRAM). Furthermore, each world is provided with a distinct MMU interface, which will enable each VM to possess a particular set of virtual memory addresses that translate to physical addresses. Interrupts may be configured to redirect to a certain world, using the Generic Interrupt Controller (GIC). By using the TrustZone Protection Controller (TZPC), system-level devices are able to be dynamically configured as one of the security states. Finally, at cache-level TrustZone adds an additional tag which represents the state in which memory was accessed.

2.2.1 Design

The LTZVisor hypervisor [PPG+17c] was designed with three fundamental principles in mind. These principles were shaped by its main design idea, which is the use of the TrustZone technology to assist the development of a virtualization layer. By exploiting this technology, this hypervisor is able to rely on hardware support to the max and at the same time mix it with a software part, as well as privileging the secure execution processor state. The principles are the following:

1. **Minimal Implementation** - By having TrustZone technology providing hardware support for virtualization, the code can be minimized, thus reducing a main cause of attacks exploited by hackers, spaghetti code. Reduction of the Trusted Computing Base (TCB) is equally procured to hold back those attacks, by statically configuring each hypervisor component;

2. **Least Privilege** - TrustZone technology provides features which enable de-prioritization of the non-secure CPU execution state, and introduces a new processor mode which brings a new level of privilege. By exploring these features, a virtualization layer that guarantees hardware isolation and different levels of priority between the two environments running on top can be implemented.

3. **Asymmetric Scheduling** - Embedded environments do not form an ideal scenario for virtualization, due to its inherent constraints on resources and timing for example. So, a different policy must be adopted in order to assure its requirements are not compromised. By privileging the execution of the secure environment above the non-secure one, an asymmetric scheduling policy, real-time characteristics of an RTOS running on the secure VM would be met.

2.2.1.1 Architecture

LTZVisor [PPG+17c] simply relies on the two security state environments provided by TrustZone technology in order to implement a virtualization solution. Figure 2.4 illustrates the proposed architecture, in which each VM is directed to one of the processor states, secure and non-secure. The secure world contains software with higher priority tasks, typically an RTOS, while the non-secure world contains software without such requirements, leaning to the GPOS side.

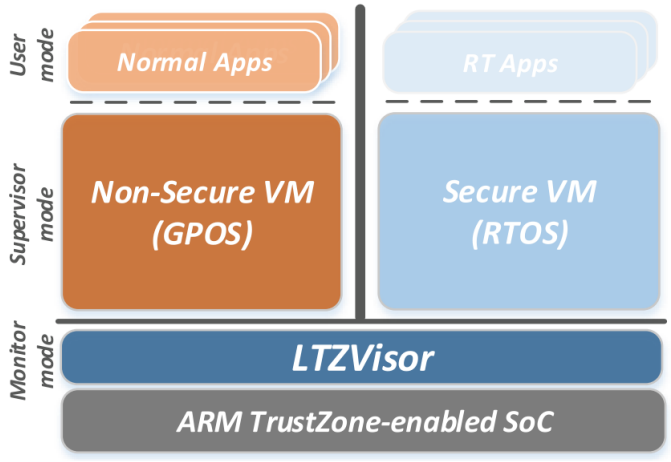


Figure 2.4: LTZVisor general architecture. Reproduced from [PPG+17c].

The monitor mode, whom provides the highest privileged processor mode, is used to accommodate LTZVisor. By having the hypervisor to run on this level, control

of the full platform is received, hardware and software wise. This way the hypervisor is able to adequately configure memory, interrupts and devices entrusted to each VM. In addition, it performs the Virtual Machine Control Block (VMCB) switches every time during a VM switching operation. During run-time, this monitor mode is called and the hypervisor transfers the active VM's state to the respective VMCB, succeeding a load of the state of the VM about to be executed to the physical CPU context.

On the secure world side is where the secure VM resides. Since this guest runs on the secure state of the processor, accesses and modifications to its state itself and its resources might compromise the hypervisor. Thus, the OSes running on top of the secure VM must be aware of the underlying virtualization layer. For this case, an RTOS running on this VM is the ideal scenario, where criticality and timing requirements are met because of the high privilege of execution provided to the VM.

On the other hand, the non-secure VM resides on the non-secure world side. In comparison to the secure VM, this one has the advantage of being completely isolated from the other software running on the secure world side, hence is able to be unaware of the underlying hypervisor. Additionally, any attempt to access and modify state information and resources from the secure world will fail and trigger an exception fault directed to the hypervisor. Forming the ideal case scenario for a GPOS to be part of the non-secure VM, running functionalities not so much time constrained, such as Human-Machine Interfaces (HMIs) and network based applications.

2.2.2 Implementation Analysis

The following sections will focus on the implementation aspects of the LTZVisor [PPG+17c]. All the details about the exploitation of the TrustZone technology are exposed, demonstrating the process behind virtualization of the CPU, how memory and devices are managed and isolated, how MMU and caches are handled and showing the procedure behind interrupt management, time isolation and inter-VM communication.

2.2.2.1 Virtual CPU

From the LTZVisor's [PPG+17c] point of view, TrustZone security extension expands a physical CPU into two virtual ones. Each of them is represented by a different security state of the processor execution, secure and non-secure. Both of the states have individual copies of some of the CPU's registers, which facilitates a virtualization scenario since part of the support is provided by the TrustZone hardware itself. Thus, the respective VMCBs of the states do not require saving and loading every processor's registers at the occurrence of any partition switching operation. The non-secure VM is formed by the subsequent registers: 13 General Purpose Registers (R0-R12), the Stack Pointer (SP), the Link Register (LR) and the Saved Program Status Register (SPSR) for each of the following modes: Supervisor, System, Abort and Undef. Similarly, secure VM's VMCB is formed by the same group of registers, except the SPSR stored is only for System mode. The remaining registers are not included because they either exclusively belong to one of the worlds or are already banked between them.

Furthermore, TrustZone states that some system-level registers are able to be modifiable only from the secure state of processor execution. From the non-secure state's side, those registers may be accessed but any attempt to modify them will be neglected. Registers as the System Control Register (SCTLR) and Auxiliary Control Register (ACTLR) serve as an example for this situation, they are used to configure memory, cache, MMU, AXI accesses, and so on. This specification provides a kind of isolation and security on the system, which benefits the hypervisor because it is the module that is able to initially configure the non-secure VMCB before booting it. However it sacrifices the possibility of the non-secure VM to configure those peripherals by itself, otherwise the VM would end in a stuck position, since it is not able to perform modifications on those registers.

2.2.2.2 Scheduler

In order to overcome scheduling issues present whenever a real-time environment is virtualized and to still guarantee its characteristics are not compromised, the LTZVisor [PPG+17c] bets on implementing an asymmetric scheduler approach, based on a idle scheduling. This policy acts in a way that assures the execution of the non-secure VM only during the idle periods of the secure VM's guest OS. But at the same time, permits the secure VM to preempt execution over the non-secure one. This is because the hypervisor prioritizes the execution of the secure

VM, which is intended to host an RTOS that will carry higher scheduling priority tasks than the non-secure VM's guest OS. Basically it means that LTZVisor is not the actual component responsible for triggering the VM scheduler, but the secure VM's guest OS itself. Despite this intentional inactivity, it is still the actual hypervisor running on the highest privilege processor mode that manages the VMCB switches, assuring at the same time that no kind of violations are performed.

2.2.2.3 Memory Partition

It is crucial that on a virtualized environment, memory gets adequately configured and segmented to belong to a certain VM. TrustZone-enabled platforms support the existence of the TZASC, which allows memory partitioning into several segments and assigning the respective security state to that portion of memory. Therefore, the LTZVisor [PPG+17c] sees this memory segmentation feature as an exploitable tool to provide spatial isolation between both VMs. By addressing each VM to its specific memory segments, configured as either secure or non-secure, the non-secure VM's guest OS will at any attempt of accessing a secure memory region fail and trigger a respective exception fault, which gets immediately handled by the hypervisor.

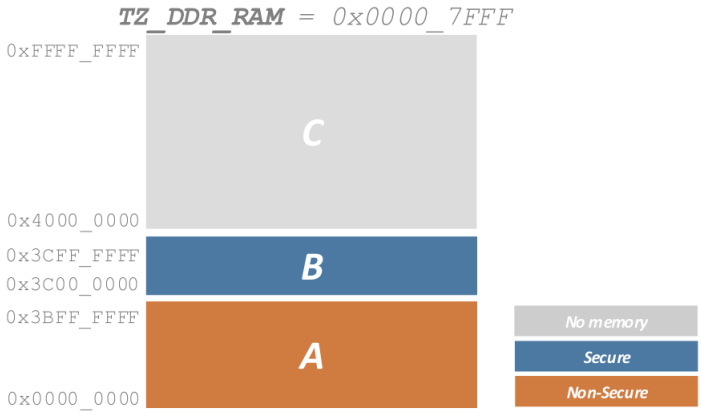


Figure 2.5: LTZVisor memory architecture. Reproduced from [PPG+17c].

Figure 2.5 illustrates the memory setup performed on the Xilinx ZC702 platform running the hypervisor. Each memory segment contains a particular bit that identifies whether this memory region is secure or non-secure. The hypervisor consists of two major memory segments, with the non-secure and secure VMs represented respectively by (A) and (B). The remaining memory represented by

(C) is not accessible. The choice of leaving the smallest chunk of memory (ranging from 0x3C000000 to 0x3FFFFFFF) to the secure VM is endorsed by the fact that RTOSes, which are the target guests for the secure VM, contain a very small memory footprint.

2.2.2.4 MMU and Cache Management

Devices which support the TrustZone security extension contain two specific MMUs for each worlds. With this, any VM can have its unique MMU configuration. As this resource is banked between the security states of the processor, performing a VM switch operation is thus facilitated since the TrustZone hardware already supports copies of the MMU's registers to represent each of the Page Table Entries (PTEs) specific to each guest OS running above the hypervisor.

Similarly, at cache level, TrustZone extends them with an additional tag, which is basically a bit that marks memory accesses whom were made from either the secure or non-secure processor state of execution. It is the hardware itself that is responsible on classifying those memory accesses present on the cache tables, relegating the software application interference regarding this, which is not able to do so. Thus, with this isolation provided by the TrustZone, cache accesses stay coherent between all the VM switches and no additional work needs to be performed by the LTZVisor, which in turn translates into slight performance advantages of the hypervisor.

2.2.2.5 Device Partition

In a related way to how TrustZone security extension allows memory partitioning, devices are able to be configured as either secure or non-secure. This actually brings the advantage of extending TrustZone security to the devices level, accomplishing isolation between the security states of the processor even to this level.

The approach adopted by the LTZVisor hypervisor [PPG+17c] is to explore this ability to classify devices in order to entrust them directly to the VMs. Along these lines, by not sharing devices between VMs, a strong isolation can thus be ensured posteriorly to a configuration of the devices performed during the boot phase of the hypervisor. The hypervisor assigns the devices to the secure and non-secure VMs by respectively configuring the devices as secure and non-secure. This way, it is ensured that accesses to secure devices from the non-secure VM fail

and will automatically trigger an exception that is immediately handled by the hypervisor, protecting the state of any device belonging to the secure VM.

2.2.2.6 Interrupt Management

The GIC is a component on ARM processors that support and manage the interrupts of the system. On those devices which include the TrustZone security extension, the GIC provides additional support for managing both the secure and non-secure interrupt sources, and at the same time enabling prioritization of secure interrupts above the others. The concept of Fast Interrupt Request (FIQ) and Interrupt Request (IRQ) is also given support by allowing to redirect them to one of the different interrupt sources.

The LTZVisor benefits of this configuration for selecting secure sourced interrupts to the FIQ group, and the remaining non-secure interrupts to the IRQ group. Initially, the hypervisor uses the GIC to define interrupts as belonging to the secure and non-secure states through the Interrupt Security Registers set (ICDISRn), and posteriorly enables the FIQ mechanism of the processor. This assures that interrupts reserved to the secure VM can be attended without added overhead, since it is the secure VM's guest OS responsibility to directly handle the interrupt. Additionally, secure interrupts are configured as the highest regarding priority of execution, thus, if an interrupt from the non-secure VM is triggered while the secure VM's guest OS executes, its execution flow is not affected, leaving the non-secure interrupt unattended until the moment the non-secure VM becomes active. The indicated prioritization is used to prevent DoS attacks that try to affect secure VM's behavior, coming from the non-secure VM. The other way around, when a secure interrupt is triggered while the non-secure VM is active, contrasts with the aforementioned situation because in this case the execution flow will directly switch to the hypervisor. At that instant, the hypervisor handles the interrupt and performs the VM switching process. Fundamentally, from the secure VM's perspective, interrupt latency is kept as minimal as possible by minimizing or removing the hypervisor's interaction. From the non-secure VM's perspective, interrupts are handled as long as its guest OS stays active.

2.2.2.7 Time Management

Typically virtualized systems provide timing services, enabling the guest OSes to be aware of the passage of time, independently. Such examples of implementing

temporal isolation include halting the time of a given partition during inactive times, or using the hypervisor to keep track of time change and updating it once the partition gets back on track.

In the interest of providing a separate time management for the VMs running above the hypervisor, the LTZVisor [PPG+17c] reserves one particular timer peripheral for each of the guest OSes. Since its configuration targets the coexistence of two OSes and the aforementioned asymmetric scheduling policy, it is crucial to not only reserve a timer for a VM, but to also configure it as a secure or non-secure device so a minimum of spatial and time isolation is guaranteed. The Triple Timer Counters (TTCs) 0 and 1 were chosen to manage the timing structures of the secure and non-secure VMs, respectively. In the context where an RTOS is hosted by the secure VM, no ticks would be missed, while at the non-secure VM hosting a GPOS in a tickless mode, passage of time would still be acquainted, as noted in the example configuration of this hypervisor.

2.2.2.8 Inter-VM Communication

An important functionality on a virtualized environment is the ability to perform communication between the VMs. The LTZVisor hypervisor [PPG+17c] opted to use a standardized approach called VirtIO [Rus08] as a transport abstraction layer.

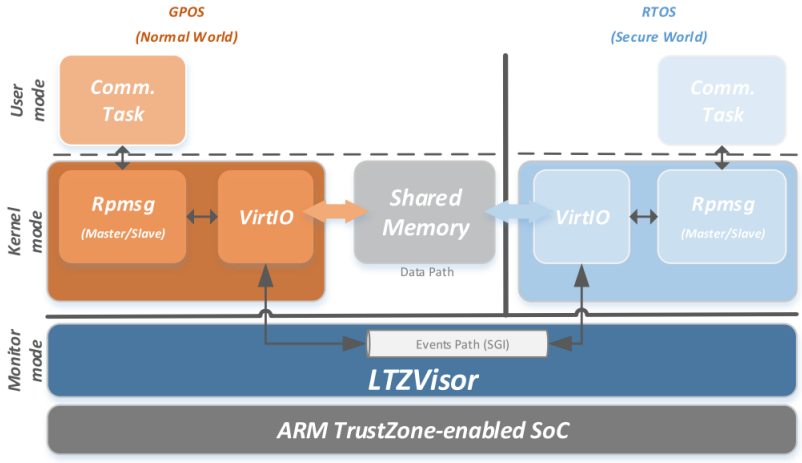


Figure 2.6: LTZVisor Inter-VM communication. Reproduced from [PPG+17c].

LTZVisor takes on the RPMsg API, which stands as a solid groundwork for communication on top of GPOSes and bare-metal approaches, and implements an

adapted version to mix both of its fundamental characteristics. Figure 2.6 depicts the communication architecture. Asynchronous communication is promoted to deal with the fact that timing requirements of the secure VM must suffer the least interference, thus, the event and data paths are separated. Technically, event paths are based on Software Generated Interrupts (SGIs), transmitted by the hypervisor running on monitor mode, similarly to how VM switching processes are performed, via an SMC instruction. A circular buffer within the shared memory block, marked on the figure, is used to store the messages in a First-In First-Out (FIFO) scheme. Then, simply put, when a VM gets back on track and has a message awaiting, the hypervisor generates a SGI to warn the respective hosted OS about the arrival of a message, whom will then proceed to access the FIFO and retrieve the message. This may cause an added overhead during a VM switching process, but compensates on guaranteeing a low-latency and reliable way to perform transaction between VMs.

2.2.3 Summary

Having done a top-down inspection of the LTZVisor TrustZone-assisted Hypervisor [PPG+17c] implementation, it is safe to say that ARM's security extensions provide an exploitable way to reliably assist on the development of virtualization techniques. This case features a minimal approach that is based on using the least amount of software as feasible, while relying on as much hardware as possible, in order to adequately satisfy typical embedded OSes' requirements and to induce the slightest overhead.

Fundamentally, this hypervisor demonstrates that the co-existence between two differently classified OSes running side by side on embedded real-time platforms is thus possible. For making the most out of the LTZVisor hypervisor, featuring an RTOS alongside a GPOS does not only prove nowadays as beneficial for combining different real-time characteristics on the same hardware platform whilst improving its wide spectrum of functionalities, but also the best way in which TrustZone's features can be embraced to collaborate on the virtualization layer.

In order to create a virtualized environment, two typical characteristics as spatial and time isolation are demanded. As seen on the previous sections regarding this hypervisor's implementation, these characteristics are fulfilled mainly by the use of TrustZone facilities to partition memory and devices. Additionally, its interrupt

management support can be perfectly oriented to satisfy the asymmetric scheduling policy. Overall, LTZVisor establishes as a reliable groundwork for a minimal virtualization layer that maintains real-time characteristics of the system, while at the same time keeping an acceptable performance for devices whom support TrustZone security extension.

2.3 ARM Architecture Overview

ARM is a Reduced Instruction Set Computer (RISC) architecture. The most widely used and popular CPUs in the market now use either the ARMv7 or ARMv8 architectures, two of the latest defined, that are able to satisfy a huge range of applications. Key benefits of the ARM architecture are implementation size, security, performance, and lower power consumption [NMB+16, ARM18a].

There are three slightly different profiles for each ARM architecture version [ARM18a, NMB+16]:

- **A-Profile** - Used in complex compute application areas, such as servers, mobile phones and automotive head units;
- **R-Profile** - Used where real-time response is required. For example, safety critical applications or those needing a deterministic response, such as medical equipment or vehicle steering, braking and signaling.
- **M-Profile** - Used for deeply-embedded chips. It is a profile that focus on key points such as energy efficiency, power consumption and size. For example, in small sensors, communication modules and smart home products.

The M-profiled ARM architecture represents huge success in the electronics industry, for being used in all ARM Cortex-M processors, the most popular processor series in this industry. These processors are available in over 3500 microcontrollers parts from most of the microcontroller vendors [Yiu15].

Over the decades, the ARM architecture has introduced new features to meet the growing demand for new functionality, better security, higher performance, and the needs of new and emerging markets [ARM18a]. Following in the next section, the focused topic will be to detail in-depth the latest M-profiled architecture version, ARMv8-M, which is the only one offering support of ARM TrustZone security extensions to microcontrollers.

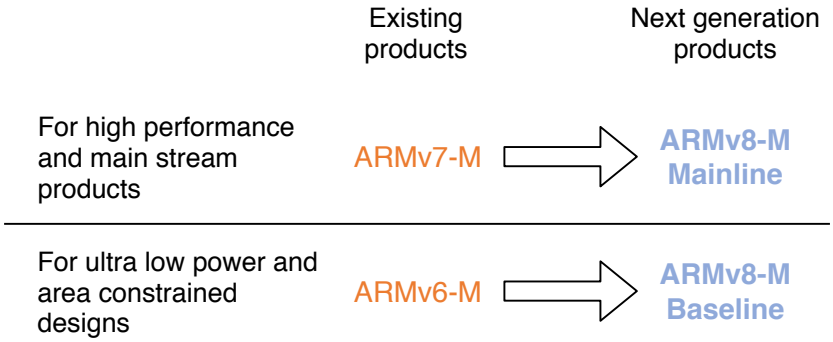


Figure 2.7: Separation of sub-profiles in ARMv8-M architecture. Adapted from [Yiu16].

2.3.1 ARMv8-M Architecture

The ARMv8-M architecture stands as a next-generation architecture for the ARMv8-M processor family of real-time deterministic embedded processors. It was built based on the success of the existing ARMv6-M and ARMv7-M architectures. This architecture remains as a 32-bit architecture and is highly compatible with the previous existing solutions, in order to enable easy migration of software within Cortex-M processor family. Security and productivity are the newest principles for every embedded solution using ARMv8-M architecture [ARM16b, Yiu15].

This architecture is divided into two sub-profiles: ARMv8-M Baseline, and; ARMv8-M Mainline (or "with Main Extension"). They respectively correspond with multiple similarities to the ARMv6-M and ARMv7-M architectures (see Figure 2.7), containing some significant enhancements at the instruction set and system features level. The Mainline version provides full features of the ARMv8-M architecture so it is ideally used for mainstream microcontroller products and high performance ESs, due to its richer instruction set to address the demands in complex data processing systems. While the Baseline version is more constrained than the other, it still is an adequate solution for the majority of embedded applications where security and low power consumption are key requirements. Application code can be easily migrated between these sub-profiles, because the ARMv8-M Baseline architecture is a subset of the ARMv8-M Mainline architecture [ARM16b, Yiu16].

Enhancements to enable better software design are included in this new ARM processor architecture, where the inclusion of the optional ARM TrustZone security extensions is the most significant one. This extension can also be referred to as ARM TrustZone technology for the ARMv8-M architecture, or simply for microcontrollers [ARM16b].

2.3.2 TrustZone: ARM Security Extensions

ARM TrustZone technology was created to enable a level of system-wide security by integrating protective measures into the ARM processors, bus fabrics and system peripheral Intellectual Properties (IPs). It allows a diverse range of secure system architectures to be implemented with minimal impact on the cost of the devices, by providing a framework that combines both integrated hardware and software components of the security extension [ARM09]. In other words, TrustZone technology is a tool that permits the addition of another dimension of security control, called secure world, allowing multiple security domains within a single processor system [ARM16b].

Focus on the establishment of a strong foundation for attaching security solutions to a platform is what TrustZone is all about. It was developed fundamentally to enable the construction of a programmable environment that allows the confidentiality and integrity of almost any asset to be protected from specific attacks [ARM16b].

Technically, the security of TrustZone is based on the idea of partitioning all of the platform's hardware and software into two worlds: secure world and non-secure/normal world. A barrier is established in order to prevent normal world's components from interfering with secure world's resources. Although, the inverse is not restricted. When running, the processor can either enter on one of those two worlds, and thus, may or may not access to certain resources. This gives both worlds the illusion that they own the processor [NMB+16]. Concretely, the idea behind the separation of the CPU between these two worlds is to be hardware enforced and to grant them uneven privileges, preventing non-secure software from interfering with secure world's resources. With this, a new level for offering protection to critical applications gets unlocked, since this technology opens opportunities to be able to secure applications and its data by constraining other (vulnerable) applications to the non-secure world's boundaries [PS18].

On ARM processors with A-profile architecture, TrustZone behaves in a way that what determines the world in which the current state of the processor execution lies on is a value of a new 33rd processor bit, or simply the non-secure bit. Additionally, an extra privileged mode is added to the architecture, the monitor mode. And, in order to enter this mode, a new privileged instruction - SMC - must be explicitly called. Hardware isolation is successfully accomplished by maintaining copies of some special registers banked between the two worlds. As for extending

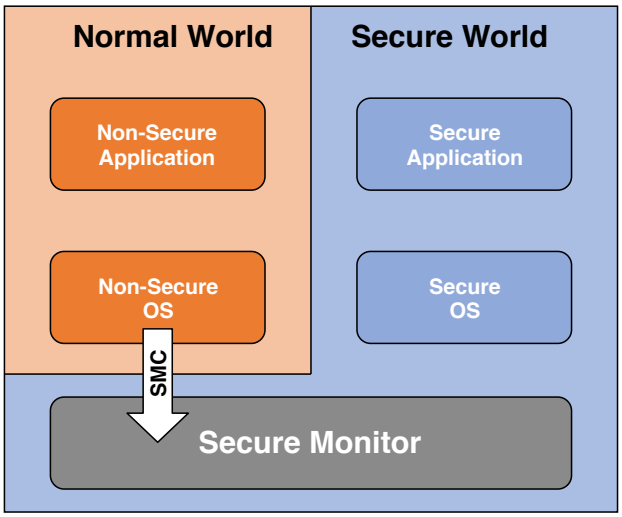


Figure 2.8: TrustZone on ARM Cortex-A. Adapted from [NMB+16].

this isolation to the memory infrastructure and peripherals, TrustZone adds the TZASC and the TZPC, whom allow memory segments and devices to be configured as either secure or non-secure. Moreover, the GIC is extended to support interrupts with different source types, enabling to redirect them to the secure and non-secure worlds. Figure 2.8 depicts the general architecture of a standard scenario on a TrustZone-enabled Cortex-A processor, demonstrating the organization of both worlds and how they are bridged using the secure monitor's software [PS18].

The introduction of this technology was well accepted by the developer community, and what started as a security extension only for the high and mid-end ARM processors ended up being adapted to cover the new generation of ARM microcontrollers (Cortex-M), with slight differences [PS18].

2.3.2.1 TrustZone for Microcontrollers

The design of TrustZone for ARMv8-M architecture is optimized for microcontrollers and low-power SoC applications. As already stated, these kind of applications require important factors such as low power, low memory and low latency. Which is why this technology is optimized to meet these requirements and has been designed from scratch instead of reusing the existing TrustZone technology for ARM Cortex-A processors [Yiu16].

Unlike TrustZone for application processors, this version of the security extension does not provide a "monitor mode". A design choice made to improve interrupt

latency, giving no need for the processor to go through an additional transition mode. Other noticeable difference relates to how the security state is determined. While on application processors this state is determined by a bit on a register (Secure Configuration Register), on ARMv8-M the process security state is determined by whether the code being executed resides in an attributed secure or non-secure memory region. Meaning that non-secure applications can directly call secure applications (running in thread or handler mode), and vice-versa [NMB+16].

Similarly to the Cortex-A TrustZone, additional security states are provided to this version: (i) secure world, and; (ii) normal world. TrustZone for Cortex-M processors follows also the principles that secure world may access all the resources, while normal world can only interact with those of which have been allocated to. As ARM Cortex-M architectures have two different processor modes, this TrustZone extension inserts security states for each. These states are orthogonal to Thread and Handler modes [NMB+16], as exemplified on Figure 2.9.

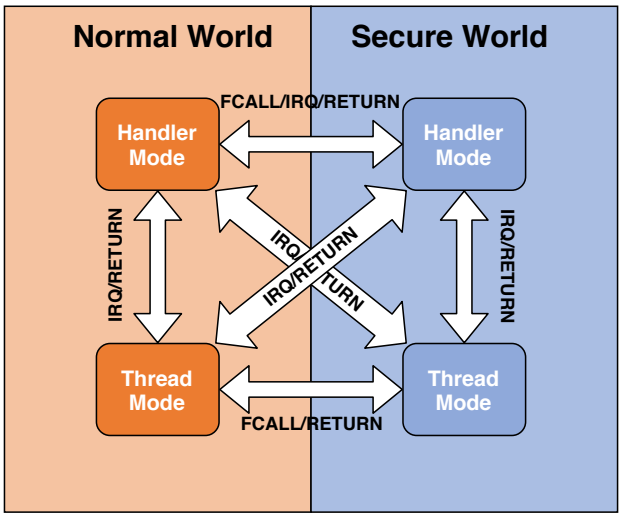


Figure 2.9: TrustZone on ARM Cortex-M. Adapted from [NMB+16].

Figure 2.9 demonstrates that secure and non-secure (normal) software can directly interact with each other either by function calls or exception handling. Functions residing in secure memory can be called from the non-secure processor state of execution, provided that the entry point for this function resides in Non-Secure Callable (NSC) memory and its first instruction is a Secure Gateway (SG) instruction. The other way around, when the processor running code on secure state intends to call a function residing on non-secure memory, is possible by using the dedicated branch instructions *BXNS* and *BLXNS* that will cause a transition from the secure to the non-secure state. As exceptions can be specified as either secure or non-secure, the security state is also prone to switch at any moment whenever

an exception is handled. If an exception or interrupt of a different state than the one that the processor is running occurs, then the processor would switch states to execute the interrupt handler before returning back to the initial state. Since register banks are shared between the two worlds, the hardware is prepared to save all registers and clear them before transition states, avoiding secure information leakage [NMB+16].

Non-secure exceptions can be de-prioritized during the configuration of the system, using the register committed to control interrupts. It is a simple feature that allows for the Secure software to take priority over the system, splitting both worlds' priorities [ARM17]. Furthermore, each general interrupt is able to be configured to redirect a particular world, the outcome of the TrustZone extensions to the Nested Vectored Interrupt Controller (NVIC). As this architecture is prepared to handle interrupts immediately, by default, no distinction is made whether it is non-secure or secure sourced. By this means, when a non-secure exception takes place during processor execution of the secure world, the information from the registers is pushed onto the stack and posteriorly erased automatically, in order to prevent information leakage [PS18].

Memory Partitioning

One of the biggest features present in this security extension is memory partitioning. In this ARMv8-M architecture, memory space is divided into Secure and Non-Secure sections, where the barrier of memory and peripheral access is built, and software code's security type is defined. For this, ARM introduced two similar mechanisms [NMB+16]:

- **Implementation Defined Attribution Unit (IDAU)** - An hardware unit closely coupled to the processor designed by the platform's developers. It is the primary unit responsible for memory partitioning. Useful for defining a fixed memory map [ARM16c];
- **Secure Attribution Unit (SAU)** - A programmable unit inside the processor. With accessibility only in Secure state, it can be used to override the settings of memory areas defined by the IDAU.

Depicted on Figure 2.10 are both of the memory attribution units. By default, ARMv8-M processors support SAU, but the same cannot be said for the IDAU, which is optional. But, whenever they are both present on the same chip, the processor follows a similar work-flow as the one present on Figure 2.10. If a certain

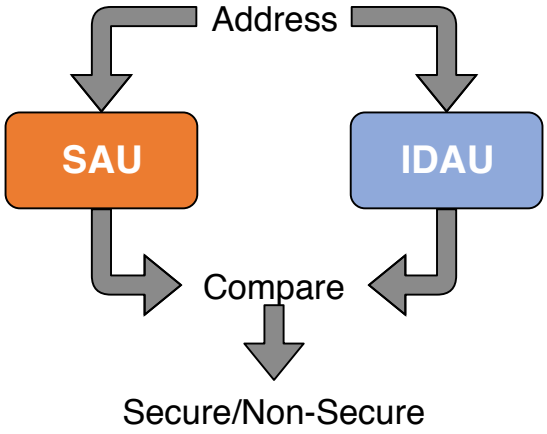


Figure 2.10: TrustZone SAU and IDAU's work-flow. Adapted from [ARM16c].

memory address that the processor tries to access belongs to a region defined with a particular attribution unit, then the respective unit checks whether or not that address belongs to a secure or non-secure memory region. If the address belongs to none of the regions defined by the attribution units, by default its considered as secure [ARM16c].

Banked Resources

TrustZone extends the internal resources across the secure and non-secure states. Inside the processor core, the banked registers are the following [ARM16c]:

- **Stack Pointers** - Separate SPs are dedicated for each secure state of the processor;
- **Interrupt Mask Registers** - PRIMASK, FAULTMASK and BASEPRI (the last one only for ARMv8-M Mainline Extension Architecture) are available for each state to perform masking of interrupts or disabling and enabling an interrupt group of a specific priority level;
- **CONTROL Register** - First two bits of this register are banked to specify each state's active SP and the current privilege level;

Beyond that, some of the internal peripherals such as the Memory Protection Unit (MPU) and the SysTick timer are also banked. They can work independently, as if they are duplicated. Additionally, a few of the System Control Block (SCB) registers are exclusive to each states, as the Vector Table Offset Register (VTOR) for example. This grants that both applications running on different security states

may have separate vector tables. Regarding accesses to the overall banked registers, the processor automatically manages to access them based on the ongoing security state. However, software running on the secure state of the processor may access the non-secure versions of the aforementioned resources using alias addresses [ARM16c].

Exception Return Mechanism

On ARMv8-M architectures which include the TrustZone security extensions, its exception return mechanism gets an added functionality. As explained previously on Section 2.3.2.1, security state of the processor might change at the occurrence of an exception originated from the opposing security state. Whenever an exception handler is entered, the processor automatically sets a special value on the LR. This value is known as *EXC_RETURN* and contains the information about which state to return and which registers need to be unstacked after handling the exception [ARM16a].

Figure 2.11: *EXC_RETURN* bit assignments. Adapted from [ARM16a].

Figure 2.11 depicts the *EXC_RETURN* register at bit-level. The most significant byte is part of a prefix, which necessarily always contains the value *0xFF*. Greyed out bits are reserved, and the rest of the bits contain the crucial info for the moment when the exception is handled. Detailedly, they are: (i) Bit 6 (S) whom indicates which of the stacks (secure or non-secure) contain the stack frame to restore; (ii) Bit 5 (DCRS) that specifies the stacking rules; (iii) Bit 4 (Ftype) that selects if the stack frame contains floating-point registers; (iv) Bit 3 (Mode) which selects the processor mode to return to; (v) Bit 2 (SPsel) that indicates the active stack pointer type, and; (vi) Bit 0 (ES) that tells whether the exception is part of the secure or non-secure world.

2.3.2.2 TrustZone-assisted Virtualization

Despite the fact that TrustZone was introduced to aid developers on security purposes, it was discovered that this technology is able to provide an hardware-based form of virtualization, a domain that was first researched by Frenzel et al.

[FLWH10]. These hardware extensions, which support a dual world execution and features such as memory and device partitioning, facilitate to contain the same number of VMs as the number of the states supported by the processor, while at the same time guaranteeing time and spatial isolation between the environments [POP+14].

With this, a scheme whereas the non-secure software runs in a bounded location like a VM that is completely managed and controlled by an hypervisor within the secure world range, is conceivable. Some TrustZone-assisted virtualization solutions of ARM's applicational processors for dual-guest [PPG+17c, SHT10] and multi-guest [PPG+17a] environments follow the particular configuration where the hypervisor is placed running on the added monitor mode, while the secure and non-secure VM's guest OSes run in supervisor mode of their states respectively. This is justified by the fact that monitor mode provides a full view of the processor, thus, by establishing the VMM there, the non-secure VM's guest OS will not require any modification. The non-secure VM will run less privileged than the hypervisor, therefore it stays completely unaware of the TrustZone extensions and what happens on the secure side. Moreover, TrustZone allows the VM switching process to be sped up, due to the overall number of the processor's banked registers among the two security states [POP+14]. Typically, a TrustZone-assisted hypervisor will follow a dual-OS approach since the number of VMs perfectly match the number of security states supported by the CPU.

TrustZone-M-assisted Virtualization Challenges

Given the previsouly detailed differences between TrustZone on Cortex-A and Cortex-M platforms, the existing TrustZone-assisted hypervisors are not directly amenable to modern Cortex-M processors. To make TrustZone-M-assisted virtualization a reality, several key challenges need to be addressed:

- TrustZone technology for ARMv8-M excludes the non-secure bit. On ARM processors with ARMv7/8-A architectures the non-secure bit is used to determine in each world the processor is executing;
- TrustZone-M excludes also the monitor mode. The monitor mode is the CPU mode used to run the hypervisor component of the existing TrustZone-assisted virtualization infrastructures.
- The Instruction Set Architecture (ISA) of TrustZone-M enabled MCUs excludes the SMC instruction. The SMC instruction is widely used in existing

TrustZone-assisted hypervisors to explicitly transition between VMs;

- The TrustZone-M specification does not include a TZASC neither a TZPC. Instead, SAU and IDAU units exist to replace them. The TZASC and TZPC are fundamental security controllers useful for memory and device partition in existing TrustZone-assisted hypervisors, and;
- The TrustZone-enabled GIC enables a clear separation and segregation of IRQs and FIQs as non-secure and secure interrupt sources, respectively. The TrustZone-enabled NVIC does not provide FIQ interrupts.

2.4 Discussion

In essence, this chapter agglomerates all the crucial elements necessary to decipher any doubt about whether or not virtualization can be used in ESs, and most importantly, in the range of devices with the lowest power and cost, the target type of processors for this thesis.

This thesis considers that recent ESs are capable of supporting multi-processed environments and at the same time providing security improvements. As virtualization had lately become a common solution in the embedded world, mainly on the implementation of hypervisors, several design forms were studied to significantly improve its use for the (restricted) embedded devices' own end, and also to reduce as many drawbacks as possible. Security improvements are the consequence of the use of virtualization, due to the isolation between VMMs it should provide.

Some example virtualization solutions were presented on section 2.1.2, those of which share similar key points and design objectives with the lLTZVisor hypervisor. Some interesting resemblances with lLTZVisor are found on SPUMONE [KYK+08] regarding its guidelines of minimal guest OS code modification, a lightweight virtualization layer and the goal to run a RTOS and a GPOS on the same platform, which motivated exploration interest. Besides, Xen on ARM [Xen18] also sparked attention since it is a solution that focuses on porting an existent hypervisor to a CPU architecture that was not yet supported, similarly to the relation between lLTZVisor and LTZVisor [PPG+17c] hypervisors. The same can be mentioned of OKL4 Microvisor [HL10], which shares design similarities to support mixture of real-time and non-real-time software on the same device, with goals of inducing minimal overhead and code size. Finally, the concept of taking

advantage of TrustZone technology for virtualization is introduced with a brief summary of some available solutions.

Furthermore, this chapter contains a section dedicated entirely to describe the LTZVisor [PPG+17c] hypervisor, by means to accurately represent its behavior and to show how similar principles can be effectively applied onto the development of the lLTZVisor hypervisor.

Newer ARM architectures for microcontrollers, as ARMv8-M, were launched and attached to it there was this new version of the TrustZone technology, whom beforehand was only present in ARM application processors. This generated interest due to the possibility of replicating the exploitation of this technology to implement an hypervisor, as done in LTZVisor [PPG+17c], but for the lower-end ARM devices. The two TrustZone versions were studied and analyzed in this chapter so the differences are enlightened for the understanding and adaptation of LTZVisor's methods.

3. Research Platform and Tools

This chapter presents the chosen platform and the tools that played a central role on the development of the lLTZVisor hypervisor. Typically there is a vast field of choice that needs to undergo a process of selection, but in this case, such a recent topic with several requirements and restrictions, demanded initially the use of simulation platforms and just later the physical development boards started to be released and become an option. As for the rest of tools, their choice fell to standardizations and suitability.

The organization of this chapter follows the indicated structure: Section 3.1 displays the platforms used to deploy and test the lLTZVisor, those of which belong to the TrustZone-enabled microcontrollers family. Then, Section 3.2 will provide information about the chosen OSes that were offered support to the lLTZVisor hypervisor. It closes with Section 3.3, along a description of the benchmark suite to test the OS's performance when running on the hypervisor.

3.1 TrustZone-enabled Microcontrollers

The Cortex-M23 and Cortex-M33 processors are the newest addition to the ARM product family of processors designed for microcontroller use. Characteristics such as real-time deterministic interrupt response, low-power, low area, ease of development and 32-bit performance are kept, specific of processors belonging to the ARM's M-profile architecture. The inclusion of the TrustZone technology to these processors was seen as an action by ARM to address new requirements, given the rising demand for IoT. These new generation Cortex-M processors were designed to fill the market need for smart and connected devices and to become the security foundation for all ESs [MSY16].

The actual microcontrollers that include this next generation of the ARM Cortex-M processors are the target for the lLTZVisor hypervisor. They are described on the following sections.

3.1.1 ARM MPS2

The ARM Microcontroller Prototyping System 2 (MPS2+) Platform is a Field-Programmable Gate Array (FPGA) development board specially designed to offer support to prototype Cortex-M-based designs. It provides an excellent environment for prototyping next generation designs, offering useful peripherals such as: PSRAM, Ethernet, touch screen, Audio, VGA, SPI and GPIO. Fundamentally, hardware and software applications for Cortex-M devices can be developed and debugged using the MPS2+ platform's solid support [ARMa].

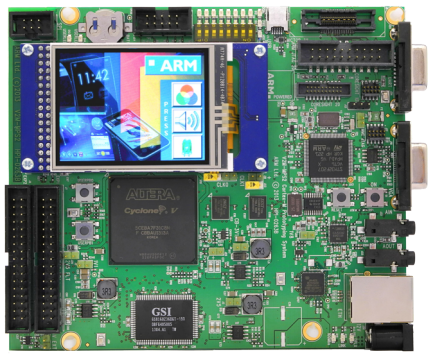


Figure 3.1: ARM V2M-MPS2 platform. Reproduced from [ARMa].

Either way, ARM offers the possibility to virtually develop on this MPS2+ platform with the introduction of Fixed Virtual Platforms (FVPs). This gives to developers the possibility to start software development without requiring a real hardware target. The simulations completely represent the ARM system model by creating a processor, memory and other peripherals within the FVP, enabling the developer to capture reliable feedback for how software will execute on a physical device [ARMc]. The ARM MPS2+ FVP was chosen to develop the lLTZVisor hypervisor ahead of hardware availability.

3.1.1.1 Fixed Virtual Platform

The Fixed Virtual Platform model implements a subset of the functionality of the MPS2+ motherboard hardware, which are enough for testing purposes. The platform can be configured as one of three different types [ARM15]:

- **V2M-MPS2 System** - The default configuration of the FVP. Has the additions of the ARMv8-M architecture and follows the ARM MPS2+ specifications. Can either use a Cortex-M23 or a Cortex-M33 processor;
- **IoT Kit** - It is similar as the previous configuration, but contains an additional IoT dedicated subsystem. Can also use either a Cortex-M23 or a Cortex-M33 processor, and;
- **CoreLink SSE-200** - Acts as an ARM CoreLink SSE-200 subsystem. It is suitable for IoT applications and contains two Cortex-M33 processors.

Essentially, all the configurations above allow achievable objectives to develop the lLTZVisor hypervisor since all provide the core of the Cortex-M23 and Cortex-M33 processors, which is what the lLTZVisor aims to develop on for providing availability to every platform that will eventually include one of these processors.

3.1.2 ARM Musca-A

The ARM Musca-A is the most recent addition to the ARM's IoT test chips and boards. It is now considered the development platform of choice for secure IoT devices [ARMb]. Figure 3.2 depicts the physical board in question.

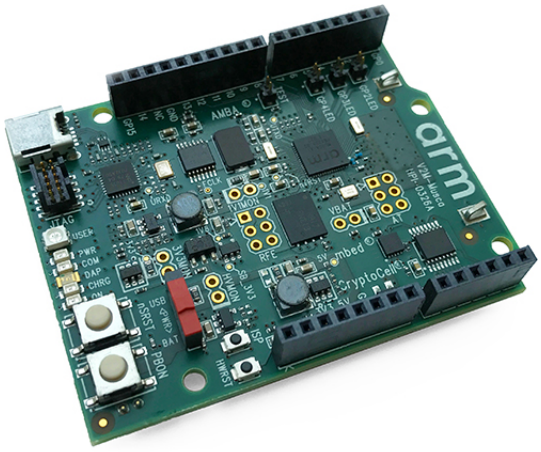


Figure 3.2: ARM Musca-A platform. Reproduced from [ARMb].

This platform was created to provide a strong foundation for developers who require security in IoT devices. It is the groundwork that will aid on designing security into products much easier, since picturing it from scratch would be time consuming and increasingly more complex [ARMb]. Fundamentally, this platform targets secure IoT designs and intends to be used as a reference implementation for other System-on-Chips (SoCs) using the same core IP.

The Musca-A test chip implements the ARM CoreLink SSE-200 subsystem, which distinctively enables design and development of a low-power and secure IoT endpoint, discussed further on Section 3.1.2.1. Sensors such as Analog-Digital Converter (ADC), Digital-Analog Converter (DAC), temperature and gyroscope are included on the board. Also, as the board interface supports Arduino's shield interface, peripherals such as GPIO, UART, SPI, I2C, I2S and PWM are provided [ARM18b].

3.1.2.1 Memory Subsystem

According to its block diagram (Figure 3.3), the Musca-A [ARM18b] encompasses the CoreLink SSE-200 subsystem featuring an asymmetric dual-core Cortex-M33, each with a private 2KB instruction cache and no data caches. The cores are connected to the main bus matrix, a multi-layer AHB5 interconnect, that enables parallel access paths between multiple masters and slaves in the system. A set of four 32KB internal SRAM (iSRAM) elements is also accessible via this main bus. Each of them features a dedicated controller and are therefore considered separate slaves on the bus. Consequently, when each of these memory elements is assigned exclusively to each CPU, it results in no contention. Although both cores can access any of these memory elements, Musca's documentation details that SRAM element 3 is a Tightly-Coupled Memory (TCM) to CPU1's data port. In contact with Arm support, it was unveiled that the remaining SRAM elements are also TCMs coupled to CPU0. This uneven coupling further increases the asymmetry of the design. Still, as part of the SSE-200, the main bus connects two slave AHB2APB bus bridges which allow access to system control registers and peripherals.

Finally, in Musca-A, two expansion ports extend the SSE-200 AHB bus matrix: one connecting a set of APB slaves encompassing I/O functionality while the other connects two memory elements targeted only to code storage and execution. The first is a 2MB code external SRAM (eSRAM) clocked at the same frequency as CPU0. Using this code SRAM for storing data is possible but impractical, as it does not support unaligned accesses. The second is a Quad Serial Peripheral Interface (QSPI) controller connected to an external QSPI 8MB boot flash memory, clocked at a much lower frequency than both CPUs. The instruction caches only cache addresses where these two memories are mapped. Although each element is accessed through distinct controllers, as explained before, they are connected to

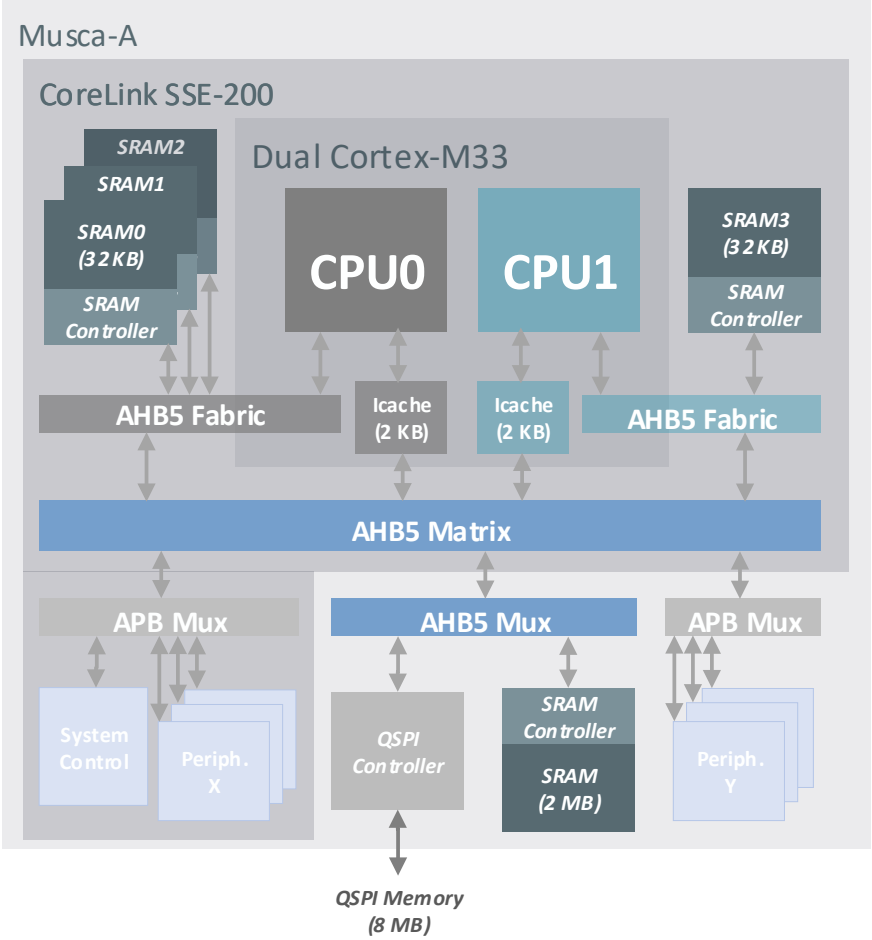


Figure 3.3: Musca-A chip memory and interconnect block diagram. Adapted from [ARM18b]

the main bus through a single expansion port for the code memory region, which prevents full concurrency when each of them is assigned to a different CPU.

3.2 Operating Systems

The majority of embedded applications nowadays use OSes, typically RTOSes. They provide a handful of mechanisms such as intertask communication, thread synchronization and deadline determinism, which facilitates the developing process of such applications. It is by this nature that the lLTZVisor hypervisor intents to support two concurrently OSes executing on the same platform, so developers can easily migrate and keep developing their applications, using a specific OS, to the newest platforms using the latest processors that benefit of the ARMv8-M architecture. Moreover, the hypervisor will provide a secure virtualization layer for the OS to run on top.

The following subsections provide a general overview about the OSes ported to the platforms previously pointed out, so that they can be used during the development of the lLTZVisor hypervisor.

3.2.1 Real-Time Operating Systems

Real-Time Operating Systems (RTOSs), as opposed to General Purpose Operating Systems (GPOSs), emphasize specially on predictability, efficiency and include features to support timing constraints. Basic scheduling support, resource management, synchronization, communication, precise timing and I/O are part of what makes an RTOS specialized to be used on highly constrained embedded devices [SR04].

Nowadays there are several RTOS options on the market, each of them with slightly different design goals, amidst certain middleware extensions and particular catalogs of platforms supported.

3.2.1.1 FreeRTOS

FreeRTOS [Fre] is a scalable real-time kernel designed specifically for small or restricted embedded devices. It was created to stand out as a simple and easy-to-use RTOS. Its source code is available publicly, provided under the MIT license. The majority of the code is written in C language. The rest is the architecturally-specific code, which is composed by a mixture of C and a few of assembly functions. This way, readability and maintainability of the code may improve, along with the propensity to port FreeRTOS to different architectures since the architecture-specific and kernel independent code belong to two different layers of the software [DGM09, Fre].

With the help of a large community behind, FreeRTOS grew into what is now today the de facto standard RTOS solution for microcontrollers. Nowadays FreeRTOS offers support to more than 35 architectures and is able to be embed in commercial products without requirements to expose proprietary source code [Fre]. Its documentation regarding API reference, combined with all the points above prove FreeRTOS as an interest case study.

FreeRTOS provides services such as: Task management, inter-task communication and synchronization, memory management, real-time events and control of input and output devices. These services are provided as a library of types and

functions within the FreeRTOS kernel's source files [DGM09]. Thus, in order to include the FreeRTOS kernel in a project, the source code has to be added do the build. Essentially, just three of the source files are needed to provide the basic functionality. The rest of the source files insert extra functionalities which are not needed for the kernel to execute, like software timers, event groups and others not mentioned on the beginning of this paragraph.

3.2.2 IoT Operating Systems

Lately, RTOSes have seen an evolution which caused them to shift the focus from just supporting safety-critical applications to those which support soft real-time applications, due to the recent technological advance of embedded devices. Such support refers to the inclusion of high-level concepts for real-time ESs, often applied to multimedia and network applications. Therefore, new ideas and paradigms started to take place in order to adapt those traditional concepts and methods to fit them into the RTOS market of today [SR04].

IoT OSes for ESs are one example of this situation. These kind of OSes aim to build network communication stacks and add some middleware support for small and restricted embedded devices, while still maintaining a slightly low-timed response to external events. Similarly to a GPOS, an IoT OS features more functionality for developers, and intends to provide a far better user experience.

3.2.2.1 Contiki

Contiki [DGV04] is an open-source IoT dedicated OS. It is designed to get the most of memory constrained systems, which are typically used in Wireless Sensor Networks (WSNs). These networks are composed by large numbers of tiny sensor devices with network capabilities that autonomously transport sensor data. On overall, devices like this are severely resource constrained because they are required to be as small as possible and cost the least reasonably amount. As Contiki intents to provide a rich enough execution environment for these devices, its mechanisms abstractions are kept lightweight in order to fit such constrained environments [DGV04].

This OS was implemented using C language and offers support to various microcontroller architectures such as Texas Instruments MSP430 and CC2538, both low-power and wireless focused solutions. As the developers of Contiki desire that

their OS reaches as many sensor device platforms as possible, since nowadays the number of these devices keep increasing, they opted to design it by abstracting the CPU from the kernel code. This code only provides basic CPU multiplexing and event handling features. Other extras of the OS are implemented as system libraries that are optionally linked with programs [DGV04].

Contiki is built around an event-driven kernel. It consists of a lightweight event scheduler that dispatches events to running processes and periodically calls processes' polling handlers. This approach is justified by the fact that using a multi-threaded model instead on such constrained environments would be pretty resource consuming. Besides, event-driven systems have been found to work well on many sensor network applications. Optionally, Contiki can combine the benefits of both event-driven systems and preemptive multi-threading, by linking the preemption system library to the application [DGV04].

3.3 Benchmarks

Benchmarks play a huge role on every project that requires some kind of performance evaluation. Via running a number of standard tests, the relative performance of a subsystem can be obtained and bring together important data that may serve as reference to measure and make trade-off decisions.

When dealing with the choice of RTOSes for an embedded application, real-time performance is generally agreed to be the most important criteria. Obviously due to key aspects on ESs such as determinism, low-latency response to external input and time critical functions. Thus, benchmark tools for RTOSes will focus on obtaining the most significant elements of performance, measuring real-time performance and comparing each RTOS to how well they perform specific critical functions. This allows the developer to quantify real-time performance and make the decision he thinks suits an application the most [Log]. The following section presents a general overview of the used benchmark suites during the development and testing stages of the lLTZVisor hypervisor.

3.3.1 Thread-Metric Benchmark Suite

The Thread-Metric Benchmark Suite [Log] is an open-source benchmark suite for measuring RTOS performance, available freely from Express Logic. As a reference,

it only provides the means to evaluate the performance of Express Logic's ThreadX RTOS [Exp]. Despite this, this benchmark tool can be easily adapted to other RTOSes by correctly mapping the new RTOS' APIs in the suite's porting layer [Log].

What this benchmark "tool" aims to assess is how fast the RTOS performs its specific functions such as: context switching, interrupt handling, communication, synchronization and memory allocation. In order for the Thread-Metric to evaluate these previous functions on multiple RTOSes for comparison, a set of common tests was conceived. These tests are coded in "vanilla" C and are based on running lots of iterations within a certain time-span. When the time-span finishes, the test restarts after the results are displayed to the developer. The selected tests focus on the following common RTOS' services: cooperative context switching, preemptive context switching, interrupt processing with and without preemption, message passing, semaphore processing and memory allocation and deallocation [Log].

3.4 Discussion

This chapter intends to gather the essential information in order to justify the choices of hardware and software resources used in the development of the lLTZVisor hypervisor, while at the same time briefly describing them.

Regarding (virtual) hardware platforms, their choice related directly to market availability. Since the technology they carry, ARM's newest Cortex-M architectures, was relatively new at the time of writing of this thesis, the lack of physical options were a constraint since all silicon vendors decided to launch their products later than the start of this thesis development. Nonetheless, the chosen platforms are still optimal choices and provide the essential to start the development of a TrustZone-assisted hypervisor. While the FVP MPS2+ virtual platform offered a way to initially test the newest Cortex-M processors and develop on them, the Musca-A test chip after being released provided a way to physically test and analyze the lLTZVisor hypervisor, an analysis that will be valid and span across many other platforms.

From an academic point of view and standardization of use, FreeRTOS came in handy to be adopted as the RTOS to run on the lLTZVisor hypervisor. The fact that it is provided freely with an open-source license and that is available on

similar architectures, boosted even more its choice. On the other side, Contiki fits like a glove to the type of OS that lLTZVisor intents to support, for the added soft real-time touch. The fact that Contiki provides a wireless communication stack, it is very lightweight and offers various libraries support make it an optimal choice for an OS that can run on constrained environments but at the same time have a little touch of a GPOS.

As for benchmarking the OSes running on the lLTZVisor, Thread-Metric Benchmark Suite proved as a complete tool for the job, not just by offering to be easily applicable to multiple RTOSes, but by evaluating every standard functionality part of a RTOS.

4. lLTZVisor: Hypervisor for low-end ARM Devices

This chapter describes the development of the hypervisor for low-end ARM devices along with every design solution chosen to overcome the challenges given by the targeted constrained environments. Section 4.1 presents the initial description of lLTZVisor, explaining in-depth what this hypervisor aims to offer. Moving forward, Section 4.2 lays out the design choices and followed principles, majorly about the TrustZone use. All implementation details for each of the hypervisor's aspects can be found on Section 4.3.

4.1 General Overview

lLTZVisor is a re-factored version of the LTZVisor [PPG+17c] targeting the recently added low-end ARM processors family (Cortex-M). It is mainly designed to achieve coexistence between two OSes on the same hardware platform, while at the same time maintaining integrity of system's security, which can easily be an attack venue due to the favorable conditions this hypervisor provides for expanding functionalities across a single platform.

The main ambition for this hypervisor is to fulfill the case scenario where two different type-oriented OSes are launched on a specific board. A typical case would be the conjunction of a Real-Time Operating System (RTOS) with an IoT Operating System (OS). lLTZVisor brings the advantage of being able to mix both of these OSes in such a way that the design principles are followed, which will be posteriorly explained in-depth on Section 4.2. Since majority of the developed real-time applications for constrained embedded devices typically have frequent idle times [PPG+17c], the IoT OS, considering it carries soft real-time requirements, will thus have a space in between those idle times from the RTOS

to be in control of the execution flow. This is exactly what lLTZVisor explores in order to provide an OS heterogeneity on a single hardware platform, to such a degree where this combination of characteristics becomes a greater alternative to reduce engineering cost and development time.

4.2 Design

The lLTZVisor's main design principle is to make use of the ARM's TrustZone security extensions to aid on the development of an hardware-assisted hypervisor targeting the newest lower-end ARM CPUs. As observed on Section 2.2, TrustZone-assisted virtualization on the LTZVisor [PPG+17c] was possible, and most importantly reliable.

Furthermore, this hypervisor borrows some principles of existing TrustZone-assisted solutions such as the classical dual-OS approach, which relies on the TrustZone exploitation. The borrowed LTZVisor's principles are summarized in the following topics: [PPG+17c]:

- **Lightweight Implementation** - With the support of TrustZone technology, relegating some functions to hardware is possible, thus causing enhancements to the overall system's performance and reductions of the hypervisor's footprint. At the same time, the number of vulnerabilities decreases since the TCB gets smaller, reducing the chance of turning code into *spaghetti code*, one way that hackers exploit to try and break into systems;
- **Least Privilege** - lLTZVisor must privilege one of the OSes (the one with hard-real-time constraints) above the other, thus respecting its fundamental requirements. A target application for this hypervisor is the combination between an OS with real-time characteristics and another without. By configuring the system so that the real-time OS may access system's resources with a higher privilege than the other, or even disabling or limiting certain accesses to the non-real time OS, possibilities for this OS to compromise or affect the other's execution flow would decrease;
- **Asymmetric Scheduling Policy** - A different scheduling policy must be adopted in order to counteract timing and resource difficulties imposed by the use of virtualization on ESs. An asymmetric scheduling policy where an RTOS gets an emphasized privilege of execution above a non-real time OS fits perfectly the scenario, since the RTOS would keep its timing characteristics

and the other would remain unaffected by the de-prioritization, as hard real time capabilities are not its focus. This scheduling policy is borrowed from both SafeG [SHT10] and LTZVisor [PPG+17c].

TrustZone-based Virtualization

The TrustZone security extensions are exploited by the lLTZVisor hypervisor in such a way that both of the security states of the processor are considered two virtual environments (a secure and a non-secure VM). Respectively, they will host the privileged and non-privileged OSes.

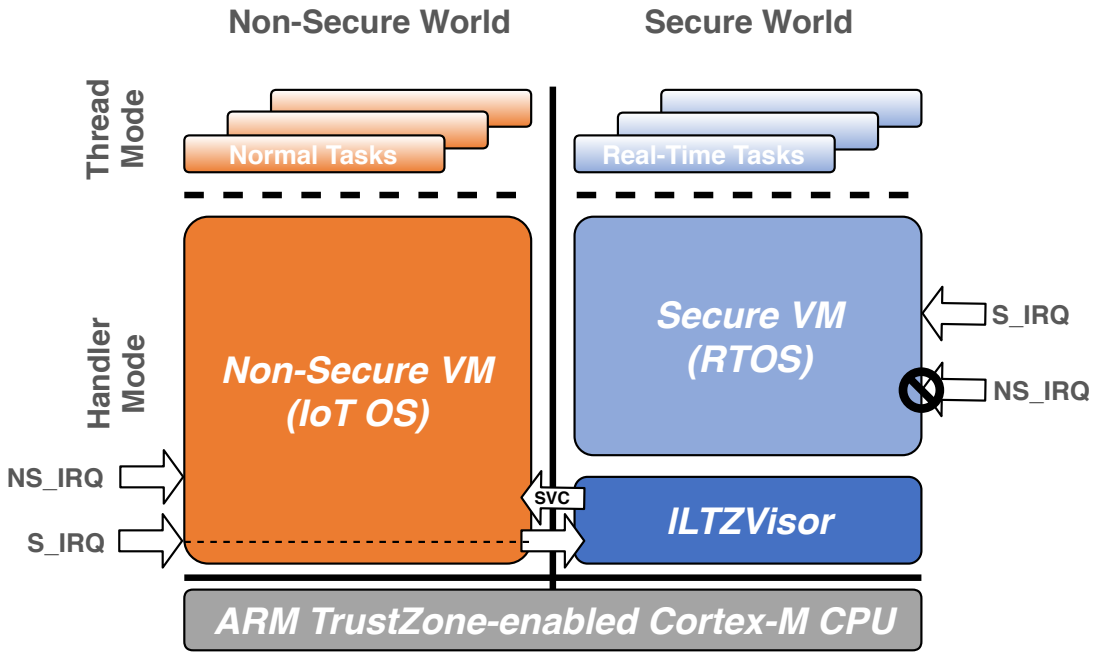


Figure 4.1: lLTZVisor general architecture.

Figure 4.1 illustrates the fundamental subsystems part of the proposed virtualization scenario. lLTZVisor shares the handler mode of the processor with the secure VM, both running on the secure world. Within this mode, the hypervisor may have full control and access of every system's resources. It is the highest privileged mode of the target processor's architecture, and thus, enables lLTZVisor to proceed with the necessary memory and devices' configuration, interrupt setup and the initial arrangement of the VMs. The nonexistence of a third privileged mode in this new variant of TrustZone (e.g., monitor mode) lead the decision to co-allocate the hypervisor with the secure VM. Although the option to de-privilege the secure VM could have been made by adopting a para-virtualization approach similar to the one proposed by R. Pan et al. [PPRP18], this decision was followed

simply based on the fact that on Cortex-A processors the secure supervisor mode has basically the same privileges as the monitor mode.

The secure VM, despite sharing the same privileged mode as the hypervisor itself, must be aware of the virtualized environment. This means that access to system's resources is available. For this reason, privileged code belonging to the secure VM, which also runs on the secure handler mode of the processor, must not in any way interfere with the other VM or modify lLTZVisor's configurations. This is a design decision made with perfect awareness, because it favors real-time and deterministic execution in favor of isolation. As represented on the Figure 4.1, the ideal hosted OS for this context is an RTOS, whose real-time requirements can be met due to the higher privileged execution given to the secure VM.

On the other side, the non-secure VM runs on the other isolated side of the processor, the non-secure world. On this processor state the hosted OS is not able to access the TrustZone's configuration registers, so no modifications to the overall system functionality is possible to be performed from the non-secure VM. Additionally, any attempt to access resources dedicated to the secure VM, will automatically trigger an exception that redirects execution to the hypervisor. Fundamentally, the hosted non-secure VM's OS runs as it would on a equivalent ARM processor without security extensions. This forms a scenario where an OS without real-time requirements could fit perfectly and provide relatively high-level functionalities in comparison to the secure VM's guest, like an IoT OS as represented on Figure 4.1 for example.

4.3 Implementation

lLTZVisor pursues a similar approach as the LTZVisor hypervisor [PPG+17c], in which the exploitation of the ARM TrustZone technology to implement a minimal virtualization layer is fulcral. Contemplating the main goal of this thesis, that is creating such hypervisor on relatively even more constrained environments, it can be assured that the implementation will have its challenges, as for instance guaranteeing that real-time characteristics of the secure VM are not put at stake.

In the following sections all the technical details about lLTZVisor's implementation are provided, kicking off on how the CPU is virtualized across the two worlds, and how these worlds are scheduled. Then, it is explained how memory and device partitioning is done, since their roles are quite relevant to the space isolation aspect

of the hypervisor. As for the time isolation aspect and how interrupts and time are managed, explanation is also present. The Chapter ends with the exposure of the multi-core AMP variation of the lLTZVisor, detailing the implementation modifications to enable the hypervisor to leverage from multi-core environments.

4.3.1 Virtual CPU

At the hand of TrustZone security extensions, some of the CPU's registers are banked between worlds. From the lLTZVisor perspective, this feature can be seen as a great way to help on diminishing the overhead associated with hypothetically having to manually save and restore all the general-purpose and special registers at every execution swap of the VMs. Figure 4.2 depicts the banked and non-banked registers of the ARMv8-M architecture.

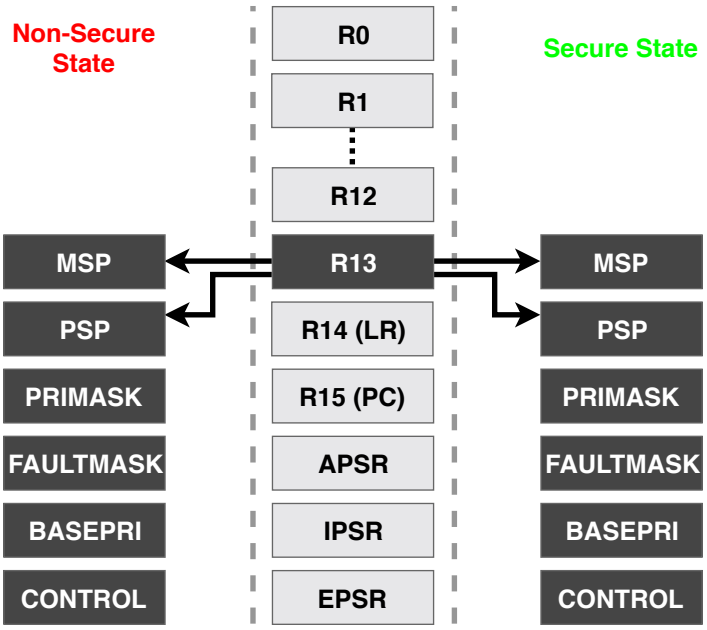


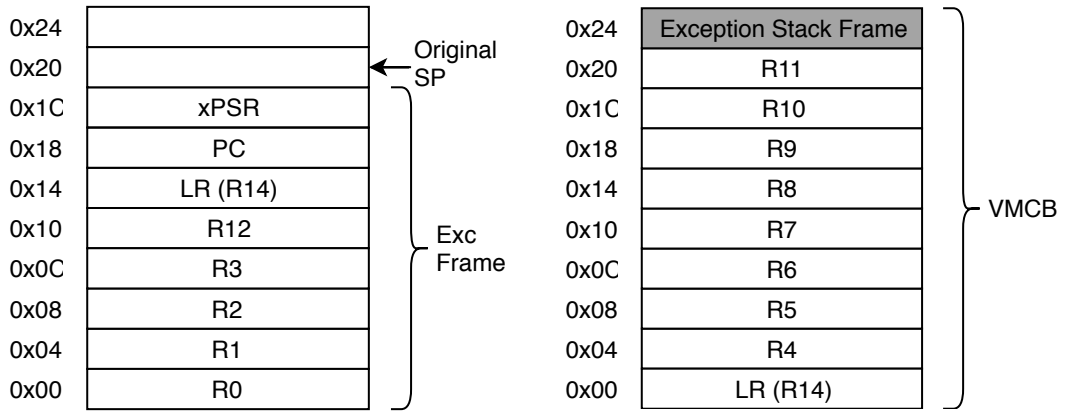
Figure 4.2: ARMv8-M secure and non-secure registers. Reproduced from [ARM16b].

As illustrated previously, all general-purpose registers, which range from R0 to R12, along with some of the special-purpose ones, are not banked between the two worlds. They are the following: Link Register (R14); Program Counter (R15), and; Application, Interrupt and Execution Program Status Registers (respectively APSR, IPSR and EPSR). Considering that these registers are not offered support by the TrustZone technology's hardware, as opposed to the Stack Pointer (R13), PRIMASK, CONTROL, FAULTMASK and BASEPRI registers which are granted

with an individual copy on each of the worlds, the VMCB will have to contain all the un-banked registers.

VMCBs are managed at every world switch, and it is the sole responsibility of the hypervisor to sanitize any sensitive information held in these registers. That being said, a parcel of the isolation is at least covered by the TrustZone hardware itself, providing copies of some important registers within the two execution states. However, based on the understanding of the Cortex-M architecture, there are still possible optimizations to the VMCBs in order to reduce the amount of registers manually saved and restored.

Upon the occurrence of an exception, ARM processors with the ARMv8-M architecture follow the standard exception stack frame model defined by the Procedure Call Standard for the ARM Architecture (AAPCS), which is depicted on Figure 4.3a. By adopting the use of exceptions to support world transitions, the majority of non-banked core registers are automatically (un)stacked by the hardware itself, which significantly reduces the world switch overhead.



(a) ARMv8-M exception stack frame model. **(b)** lLTZVisor VMCB.

Figure 4.3: Stack frames comparison.

Following this scheme, the only registers that will require a place on the VMCB (depicted on Figure 4.3b) are the general-purpose registers ranging from R4 to R11 inclusive and Link Register (R14) as well (explained in the following subsection), whom will be saved and restored manually using the same stack used by the processor to produce the exception frame ("Original SP" depicted on Figure 4.3a). Consequently, since the stacks are not shared between worlds and given their important role of storing the respective VMCB of each world, additional memory reservation will not be required, which is great for constrained environments where memory is not abundant. This design choice tries to squeeze out the most of the

ARMv8-M architecture, by relying as much as possible on the hardware in order to keep lLTZVisor as most lightweight as possible, starting with a reduced VMCB.

4.3.2 Scheduler

The scheduling policy strictly follows the LTZVisor hypervisor [PPG+17c] and SafeG's [SHT10] strategy. This policy prioritizes the secure VM's execution above the non-secure's one, an asymmetric scheduling policy or simply idle scheduling. Right off the bat, this strategy overcomes the threat of not jeopardizing the real-time requirements strictly mandatory by the RTOS on the secure world. After all, this policy means that the non-secure VM is only scheduled during idle periods of the secure VM. Moreover, when the non-secure VM is executing, the secure VM can at any moment preempt and take control of the execution. These transitions will occur by means of a triggered exception, the RTOS' tick or some other external interrupt.

The TrustZone technology provides a system level register named Application Interrupt and Reset Control Register (AIRCR), which contains a bit field that allows de-prioritization of non-secure configurable exceptions to enable secure exceptions to take priority (AIRCR.PRIS), exactly what lLTZVisor aims for. This configuration is performed during the system's boot to ensure that non-secure exceptions are de-prioritized. A transition from the secure VM to the non-secure VM is handled by the hypervisor when the secure VM's guest OS reaches an idle state, and thus, issues a Supervisor Call (SVC). However, this call must not be mistaken by a regular OS operation, since both the lLTZVisor and the secure VM share the SVC handler. Therefore, any SVC results in a trap to the hypervisor, which will have to differentiate ordinary OS operations from calls to trigger VM transitions.

Algorithm 1 Schedule non-secure VM.

```
switch secure vector table;
save secure vm's registers;
load non-secure vm's registers;
enable non-secure exceptions;
```

The Algorithm 1 is part of the SVC exception handler, thus, has privileged access and can be explicitly called by the following instruction: *svc 0xff*. Its purpose is to swap VM execution to the non-secure world. It starts by switching the secure world's vector table to a customized one belonging to lLTZVisor, which assures that every secure exception (excluding Reset and Fault Handlers) is redirected

to an exception handler containing Algorithm 2, responsible for scheduling the secure VM. After this, the secure VMCB is stored and the non-secure VMCB is loaded. The Link Register (R14) is also part of these VMCB operations, since it is the register that makes redirecting the execution to the non-secure world possible, rather than returning to the secure world whom made the SVC. Before exiting the exception handler and (re-)starting the non-secure VM's execution, its exceptions are enabled since they get disabled whenever the non-secure world is not active to prevent non-secure exceptions to preempt regular execution of the secure VM.

Algorithm 2 Schedule secure VM.

```
disable secure interrupts;
disable non-secure interrupts;
switch back secure vector table to origins;
save non-secure vm's registers;
load secure vm's registers;
get current active interrupt number;
jump to active interrupt handler;
```

Similarly to the Algorithm 1, the Algorithm 2 is contained within an exception handler with privileged access. But instead, this routine is used to schedule back to the secure VM and is multiplexed by all secure exception handlers comprised on the switched secure vector table during Algorithm 1. The routine 2 starts by immediately disabling secure and non-secure interrupts to prevent exception tail-chaining. Then, replacement of the secure world's vector table back to its original place occurs. Posteriorly, after VMCB switch is made, the exception that preempted the non-secure VM to schedule the secure VM must be handled. The exception handler's address is fetched from the vector table and then executed by performing a jump to the respective address. This achieves the VM switch process and the secure VM gets back on trail.

4.3.3 Memory and Device Partition

When dealing with ARMv8-M microcontrollers that feature the TrustZone security extension, it is established that each memory address will have a security state assigned. What defines this are two units that are part of the microcontroller, which are the Secure Attribution Unit (SAU) and the Implementation Defined Attribution Unit (IDAU). As the name of this last unit mentions, it is implementation defined, which then cannot be as useful as the SAU in terms of reconfigurability during run-time of the processor. So, what was designed for configuring memory

address spaces and peripherals as secure, non-secure and non-secure callable, can at the same time be adopted to delineate both the memory regions and peripherals of each VM.

From the point of view of the lLTZVisor hypervisor, the SAU is seen as a tool seclude memory, guaranteeing spatial isolation between the two VMs executing on opposing security states of the processor. Ignoring the non-secure callable regions that the SAU can configure as, if the platform's memory gets partitioned in two, with one of them being secure and the other as non-secure, the two VMs should have by then its space for execution reserved. Although assuring that the non-secure VM cannot interfere with the secure's VM resources, the same can not be said the other way around. Only accesses from the non-secure to the secure world automatically trigger exception faults, meaning that the secure VM must be aware of the presence of another VM on the platform in order to avoid possible interferences.

The same principle regarding memory can also be applied to devices as well, since they are memory mapped. Isolation at the device level is what lLTZVisor strives for. In order to keep the virtualization layer as simple and minimal as possible, SAU hands out a perfect way to assign a particular set of devices to the non-secure VM and another to the secure VM. This unleashes the burden of the hypervisor to carry out the device partitioning itself, relying instead on the SAU unit to do that process. Accesses from non-secure to secure world's devices will automatically cause an exception and hand over the execution to the lLTZVisor fault handler.

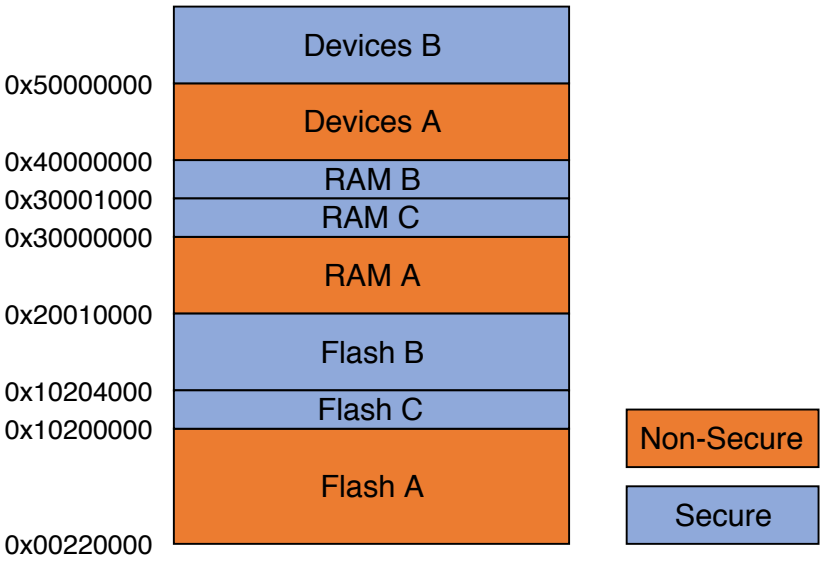


Figure 4.4: lLTZVisor memory and device configuration on Musca-A board.

During boot-time, the hypervisor configures the memory regions for the VMs and assigns their respective devices. An example configuration would be as depicted on Figure 4.4. That illustrates the memory layout configuration of the lLTZVisor hypervisor on the ARM Musca-A platform. The major three depicted components are Flash, Random Access Memory (RAM) and Devices. The first two were sliced into three parts, while the last one only into two, representatively by A, B and C. They respectively correspond to segments reserved for the non-secure VM, secure VM and lLTZVisor itself.

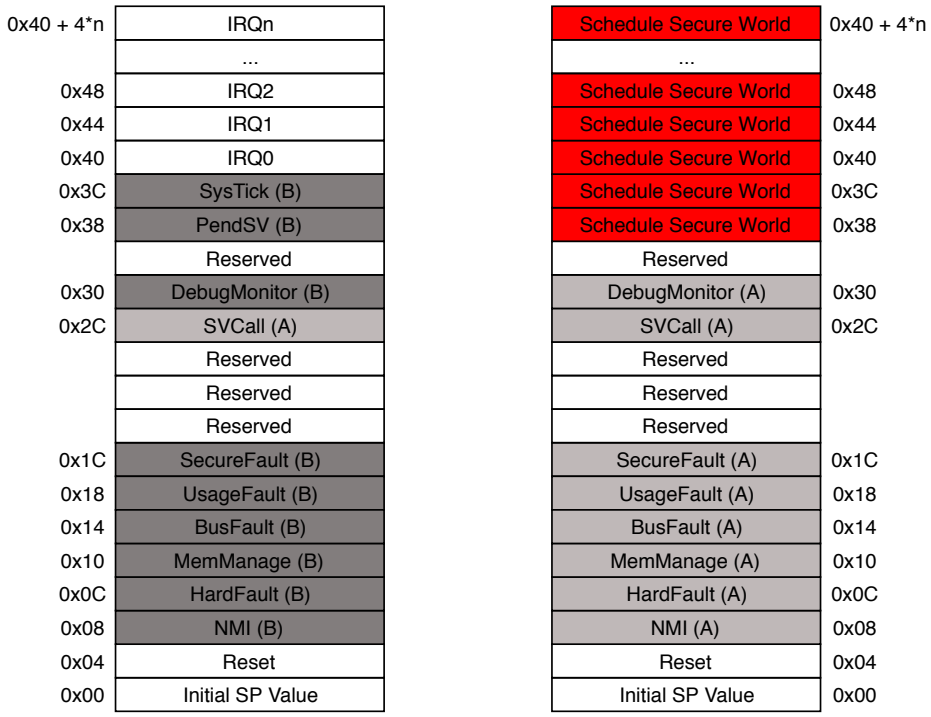
4.3.4 Exception Management

As already mentioned, the ARMv8-M TrustZone extension lacks the additional monitor mode present on the ARM's A-profiled architecture version. On section 4.3.2 a workaround is demonstrated, which is basically benefiting from the automatic processor state switches caused by exceptions, with the assistance of technically using the SVC instruction to simulate a type of hypervisor call that triggers a VM switch. Therefore, this section gives a more insightful view regarding how lLTZVisor manages the exceptions and how it structures the underlying base that allows all the VM execution swaps.

Features of the ARMv8-M TrustZone technology include banking of some internal resources, like for example the VTOR, that allows the existence of two vector tables, one for each processor state. Thus, each of the VMs running above the lLTZVisor can have their own independent exception handling. With the added value that the Interrupt Service Routines (ISRs) can be configured individually as either secure or non-secure through the *NVIC_ITNSn* array of registers, this means that a set of exception handlers can be specifically targeted to one of the processor states, or in other words, to one of the VMs.

In order for the lLTZVisor to manage the VM execution swaps by means of exception handling and depending on each VM to be scheduled, two exception vector tables for the secure world had to be created. ARMv8-M architecture permits this approach, since the VTOR can be updated on-the-fly.

Figure 4.5 depicts the two vector tables associated with the lLTZVisor's VM switching management. Within these tables, there are three colored blocks which determine different things as followed:



(a) Active Secure Vector Table. **(b)** Passive Secure Vector Table.

Figure 4.5: lLTZVisor vector tables.

- **Light Grey or (A)** - Such blocks represent exception handlers located on the hypervisor's memory segment, by other words, are part of the lLTZVisor's source code;
- **Dark Grey or (B)** - Such blocks represent that the exception handlers associated with them are handled by the secure VM directly, and;
- **Red** - Such blocks represent the exceptions that when triggered, will generate a VM switching process from the non-secure to secure processor state.

Specifically, the "Active Secure Vector Table" on Figure 4.5a is somehow a copy of the secure VM's, but contains a different handler for the Supervisor Call (SVC), which is handled by the hypervisor instead of the secure VM. This option was made to deliberately fake a monitor call, which ARMv8-M does not support, whenever the instruction *svc 0xff* was explictly triggered by the secure VM. In case the value used is different than *0xff*, this handler calls the secure VM's SVC handler. This handler managed by the hypervisor is responsible to call the function that will start the VM switching process to non-secure state. Algorithm 3 states the process described previously.

Back to the scheduling process, it was said in section 4.3.2 that whenever a VM execution swap occurred, the hypervisor would also swap the secure vector tables.

Algorithm 3 lLTZVisor's SVC Handler.

```
if Main Stack Pointer used by caller then
  get Main Stack Pointer's address;
else
  get Process Stack Pointer's address;
end if
locate stacked PC value on exception frame;
get the SVC instruction's opcode;
extract immediate value;
if immediate value is not equal to 0xFF then
  get secure VM's SVC Handler address;
  jump to the secure VM's SVC Handler;
else
  jump to function for scheduling to non-secure VM
end if
```

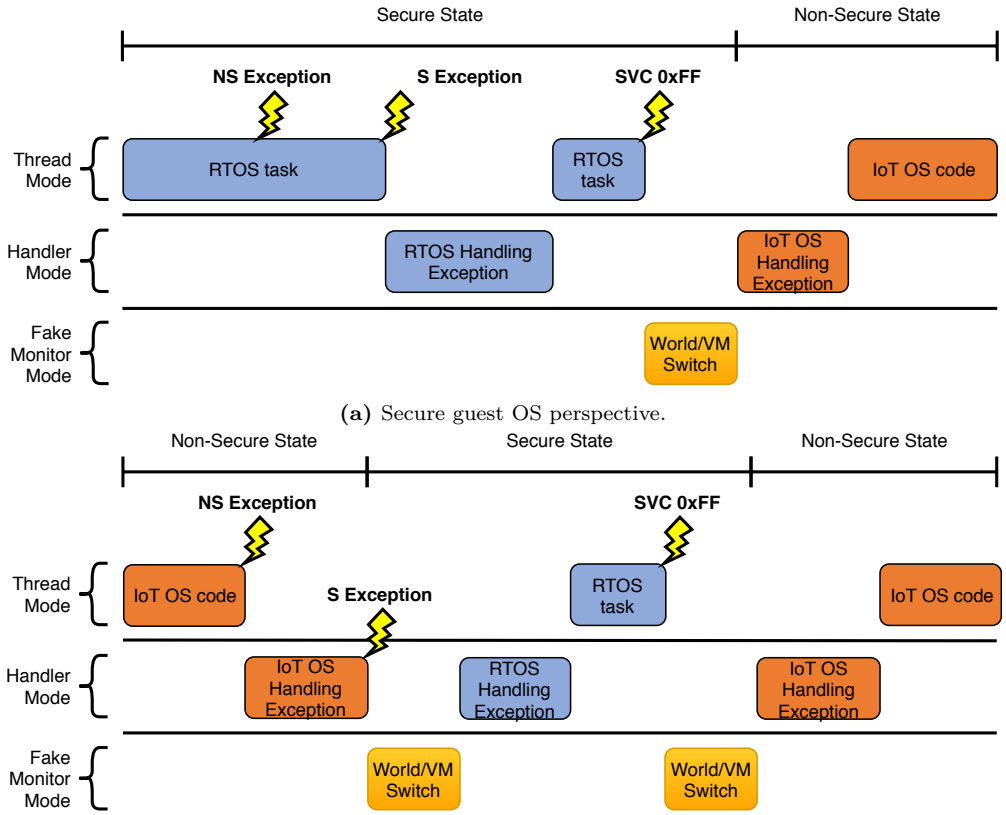
The moment the non-secure VM starts executing is when the "Passive Secure Vector Table" (see Figure 4.5b) takes the lead. Its exception handlers of the ISRs and system handlers (SysTick and PendSV) were modified so they redirect to the routine present on the lLTZVisor which schedules the execution back to the secure VM. The remaining fault handlers simply redirect to the lLTZVisor's fault handlers and halt execution.

Although this management was needed from the secure world point of view to perfectly perform the VM switches, the non-secure world is completely unrelated and unaware of this vector table transitions, due primarily to the strong isolation and the banked vector table resources offered by TrustZone technology.

Moreover, the control lLTZVisor's strives to achieve is that the secure VM or RTOS partition gets the highest priority possible and is able to preempt the non-secure VM at the occurrence of any event redirected to itself. And, as said on section 4.3.2, TrustZone permits this by de-prioritizing the non-secure world's exceptions. Which again is also perfect considering that no additional code or modifications have to be made, with TrustZone providing an environment where secure world's exceptions will always upstage non-secure one's.

To sum everything up, Figure 4.6 depicts how interrupts are managed and how the particular processor states and modes transit from the perspective of each guest OS. As illustrated, no non-secure exception can preempt the RTOS execution, but only the explicit *svc 0xff* instruction can, which will trigger the fake monitor mode already explained. Besides non-secure VM's exceptions being configured to have lower priority relatively to the secure VM and are disabled while the secure

VM is running. Any attempt from the non-secure VM to change any secure NVIC register will have no effect, and any attempt from the non-secure VM to redirect an exception source to a secure exception handler will be trapped to the hypervisor. This arrangement avoids possible DoS attacks coming from the IoT OS. Likewise, whenever a secure exception happens during non-secure processor state, the execution flow is promptly redirected to the RTOS, after the lLTZVisor deals with the VM switch process. After that, the IoT OS can only return to execute where it stopped succeeding another SVC call to switch VMs.



(a) Secure guest OS perspective.

(b) Non-secure guest OS perspective.

Figure 4.6: lLTZVisor exception managament timeline.

4.3.5 Time Management

On ARMv8-M processors with TrustZone security extensions the system timer (SysTick) is banked between secure and non-secure states, which enables each VM to manage its independent system timer. This 24-bit timer is usually integrated within the core of the ARM Cortex-M processors, and is for the most part used by OSes to facilitate porting from another platforms, not requiring the OSes to adapt for supporting another specific timer peripheral on a certain platform.

With the assistance of the TrustZone security extension, lLTZVisor can follow the policy of asymmetric scheduling, without added intellectual property. An heterogeneous OS configuration is the ideal scenario for this hypervisor, where the respective VMs can take advantage of a distinctive time management approach provided by the two independent system timers. Since the secure VM running on lLTZVisor does not miss a single system tick interrupt, an RTOS can fit perfectly without having the risk of compromising its real-time characteristics. However, the non-secure VM has its exceptions disabled when not executing, thus causing ticks to be missed, so the recommended choice in this case should fall on a tickless OS, in the way that it could still execute consistently despite losing flow of the time.

4.3.6 AMP Variation

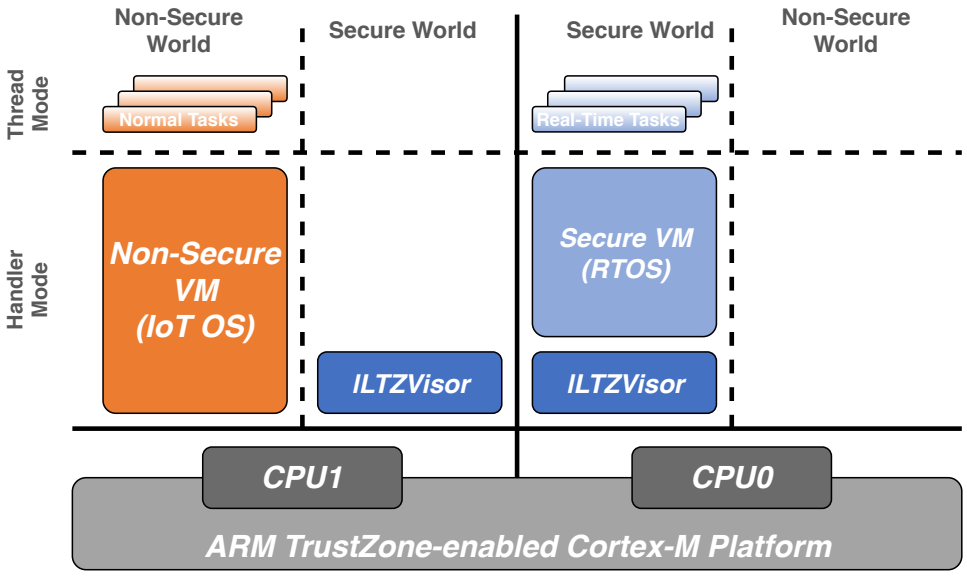


Figure 4.7: lLTZVisor AMP architecture.

lLTZVisor also offers support to dual-core processor platforms, where it implements the dual-OS configuration in an Asymmetric Multiprocessing (AMP) environment. This scheme places the VMs running concurrently on each of CPU's cores. The secure VM is assigned to the primary core (CPU0), while the non-secure VM is assigned to the secondary core (CPU1), as depicted in Figure 4.7. The hypervisor is split into two parts, one for each CPU. Both are used to perform the boot-up initializations and require no runtime roles, on contrary to the single-core approach, which needs to perform VM switching operations. This is

due to the fact that there is a one-to-one mapping between the number of guests and the number of cores.

This approach does not contain any major difference relatively to the single-core version referred on the previous sections, except a slight change on the exception management, specifically regarding the two secure vector tables swapping, which will be explained next.

4.3.6.1 Exception Management

Hitherto it was described lLTZVisor as using two secure vector tables, one of which called "Active Secure Vector Table" and another called "Passive Secure Vector Table". They would switch and control the secure state's exception handling at turns, synchronized to the occurrence of every VM swap process, in order to allow redirecting some exception handlers to portions of code that triggered a VM switch and changed the processor state. However, an AMP environment will not require such management considering that no VM switches will be performed within the same CPU core. What this approach does instead is dedicate only one secure vector table to each of the cores.

lLTZVisor contains a vector table dedicated for the CPU1. This table takes control of all the fault and system handlers of the secure state. The reset handler along with the SVC handler are used respectively to perform the initial setup of the processor with the hypervisor's configurations and to promptly jump the execution state to non-secure so the non-secure VM starts executing.

Algorithm 4 CPU1 start non-secure VM.

```
get non-secure main stack pointer;
pop LR from the non-secure main stack pointer;
exit handler;
```

After CPU1 wakes up and executes the respective reset handler, it traps the execution by using the SVC instruction. This triggers its correspondent exception handler, composed by Algorithm 4, that is used to switch the CPU1 to the non-secure state. At the point when the SVC handler executes, the non-secure world's Main Stack Pointer (MSP) is already correctly set, and contains a "fake" exception frame that redirects execution to the beginning of the non-secure VM's code. Besides this exception frame, *EXC_RETURN*, the value that indicates which processor state to jump after leaving the handler, is contained in the non-secure MSP. It is for this particular reason that on Algorithm 4 the non-secure MSP is

used to retrieve the *EXC_RETURN* value, which posteriorly is copied to the LR register. To kick-off the non-secure VM's execution, the LR register is copied to the Program Counter (PC) register, knowing that on ARM Cortex-M processors exceptions are exited by copying the *EXC_RETURN* value to the PC register.

4.4 Single-Core Execution Flow

The processor always starts on secure state. The lLTZVisor takes control instantly and performs initializations of the C runtime environment and the exception vector table. After that, the SAU unit is used to assign memory regions and peripheral segments as either secure or non-secure. IRQs are routed to either one of the worlds using the *NVIC_ITNSn* group of registers. And finally, system control level registers are configured for prioritizing secure world's interrupts above the non-secure's (using the AIRCR register on the SCB peripheral).

Algorithm 5 Fake the non-secure world's exception stack frame.

```
get ns world msp top;
load initial exc_return value to register;
load lltzvisor fault handler address to register;
load ns vm's reset handler address to register;
load initial xpsr value to register;
push the previous registers to ns msp stack;
set new ns msp stack top;
```

After the previous explained initial setup, the VMs will be prepared for execution. Since the VM swaps will be performed by exception handling, the non-secure VM will need an existent exception stack frame on its stack. So, lLTZVisor performs an artificial stacking of the registers with the values necessary to simulate an exception return to the beginning of the non-secure VM's code, as presented on Algorithm 5. The code begins by fetching the non-secure vector table's address in order to obtain the MSP top and reset handler addresses. Initial values for the *EXC_RETURN* and xPSR register are prepared for respectively performing the jump to non-secure state and resetting processor flags to an initial state. A fault handler address is also loaded, to complete the exception stack frame. These values are then pushed onto the non-secure MSP stack and form a fake exception frame to switch processor execution to the non-secure state on the beginning of the non-secure VM's code after the first request to swap VMs. The secure VM preparation is unrelated to the previous process, since the only thing the lLTZVisor needs to obtain is its reset handler address to trigger its execution.

On the next phase, lLTZVisor boots and hands out the execution to the secure VM. Once the RTOS running on the secure VM reaches an idle state, the *svc 0xff* instruction is issued and enforces the execution of an exception handler that will start the VM transition process, described on Algorithms 1 and 3. Before exiting the exception and handing out the execution to the non-secure VM, the vector tables are switched, secure state VMCB is stored and non-secure VMCB is loaded and non-secure exceptions are activated.

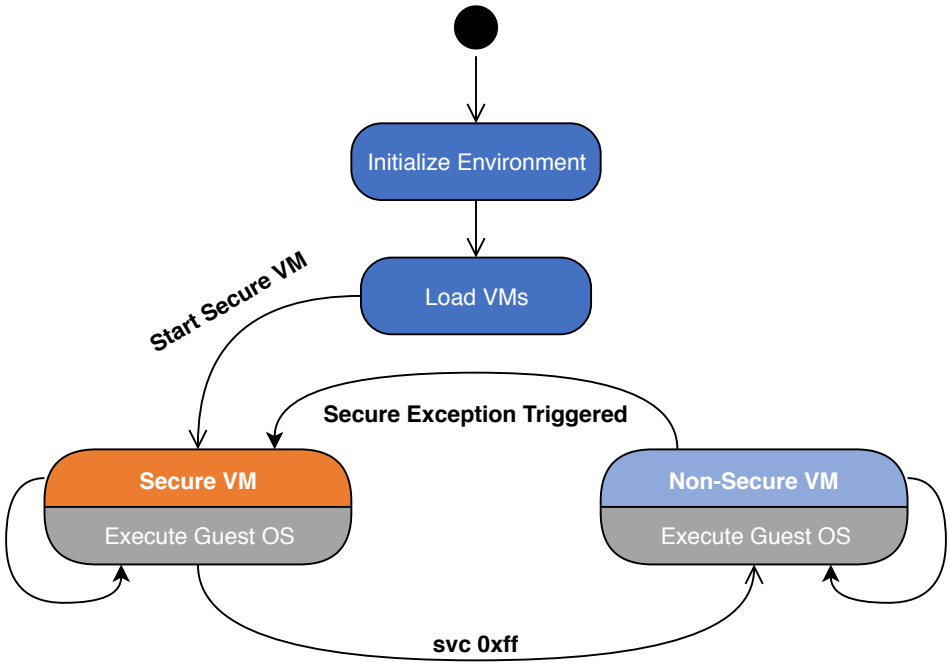


Figure 4.8: lLTZVisor Single-Core execution flow state diagram.

The moment the non-secure VM is active, its execution pursues the same path as if it would by running directly above an ARM Cortex-M processor without security extension implemented and remains completely unaware about the existence of an underlying hypervisor. The only difference is that this VM's execution stalls whenever a secure exception is triggered, because of the lLTZVisor's prioritization of the secure VM above the non-secure VM. Assuming the occurrence of a secure exception on this situation, the processor jumps to the secure state and executes the respective exception handler of the "Passive Secure Vector Table" (see Figure 4.5b) trapped at the level of lLTZVisor. If the exception raised in this case is of fault type, then the system execution is halted by the lLTZVisor fault handler. If otherwise, and the exceptions correspond either to an IRQ or of a system type exception, then the procedure to schedule back to the secure VM is executed as described on Algorithm 2. Essentially, the "Active Secure Vector Table" gets back in control, non-secure exceptions are disabled, VMCB switches occur and

most importantly, the real active exception is called and thus, the control of the processor execution is immediately handed out to the secure VM.

Freshly after the VM switch process back to the secure world, the OS inside the secure VM will resume execution as it was intended to in first place, until the idle state is reached once again, repeating the VM switching process indefinitely. Summed up, Figure 4.8 represents the overall execution flow of the lLTZVisor described earlier.

4.5 AMP Execution Flow

The AMP approach of the lLTZVisor, contrasting to the single-core approach, presents changes on the exception management behind the VM switching processes, as referred on section 4.3.6. Those changes are mainly caused due to the intended scenario of the AMP lLTZVisor, which is to assign each of the VMs to one of the cores. This means that VM switching processes are abolished and replaced with an initial scenario configuration for CPU1 by exception handling.

Similarly to the single-core approach, initializations of the run-time environment, setup of vector table, delineation of the memory regions, assignment of interrupts to each of the VMs and configuration of system level registers are made for both cores. With the exception that CPU1 initialization does not tamper with run-time environment and vector table points. Respectively because the environment is common to both cores thus should not be re-configured and because the vector table will be configured by CPU0 before waking up CPU1 for the initial boot-up configuration.

Algorithm 6 Wake-up CPU1.

```
set cpu1 vector table;
kick-off cpu1 execution;
```

The key difference contained on this AMP approach is then the addition of a wake-up call to the secondary core (CPU1) which is initially de-activated, before the primary core (CPU0) starts the secure VM execution (see Algorithm 6). It is important that a vector table is set for CPU1, because it is there where the address of the code to be executed from boot is defined, on the reset handler. By assigning CPU1 a vector table before initiating its execution, the hypervisor is able to redirect the reset handler to the Algorithm 7, which performs the initial

configurations and executes an SVC call, which also has its handler redirected to Algorithm 4 that performs the jump to the non-secure world.

Algorithm 7 CPU1 reset handler.

```
outline memory regions;
redirect interrupts to each processor state;
de-prioritize non-secure exceptions;
get non-secure msp;
perform fake exception stack framing on non-secure msp;
execute svc instruction;
```

At this point, both cores are executing simultaneously. CPU0 gets full control of the secure VM, but on contrary to the single-core approach, no SVC instruction will trigger a VM switching process. During this time, CPU1 performs the initial configurations as stated on Algorithm 7, which are exactly the same as the ones done on CPU0. The last three statements of the Algorithm 7 are responsible for the processor jump to the non-secure state, done in order to kick-off non-secure VM's execution. The performed non-secure MSP fake stack is simply done by reducing the top of the stack value by the number of an exception frame size, and then, the SVC handler is called so the jump to the non-secure state is accomplished after Algorithm 4 is carried out. The whole process is summarized on Figure 4.9.

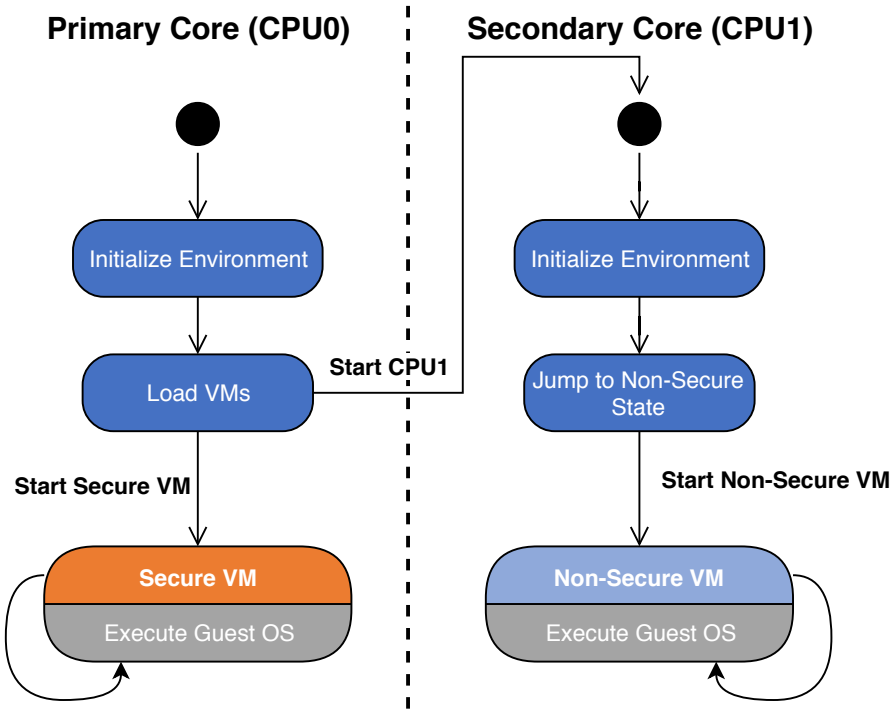


Figure 4.9: lLTZVisor AMP execution flow state diagram.

4.6 Predictable Shared Resources Management

Embedded virtualization has several proven benefits but still faces serious challenges, specially when real-time is a concern. On one hand, it already provides a reasonably high degree of time and space encapsulation and isolation of VMs by time-multiplexing resources such as the CPU, partitioning memory and assigning or emulating devices. On the other hand, partitioning and multiplexing of micro-architectural shared system resources were, until recently, neglected by most hypervisors. This led to contention and lack of truly temporal isolation, hurting determinism by increasing jitter [MBBP18, THAC18]. Also, this can be explored by a malicious VM to implement DoS attacks by increasing their consumption of a shared resource. Moreover, it allows for the existence of timing side-channels compromising data confidentiality, which might be exploited to access private or sensitive data of either a VM or the hypervisor [GYCH18, THAC18]. Although AMP hypervisors with VMs pinned to dedicated cores already remove part of this contention when compared to single-core or SMP implementations, system-wide resources such as Last-Level Cachess (LLCs), memory controllers and interconnects still remain shared and subject to contention. This is further aggravated as mechanisms such as cache replacement, cache coherency, hardware prefetching or memory controller scheduling focus mainly on performance and bandwidth maximization.

Many approaches, like cache coloring [KR17, XTP+17], memory bandwidth reservations [CBS+17], or both [MBBP18] have already been applied to mitigate these issues with promising results. However, these techniques depend on the existence of memory virtualization infrastructure or performance monitoring features which are not available on MCUs. On the positive side, many of these contention points, such as data caches or Translation Lookaside Buffers (TLBs), are seldom, if ever, featured in low-end platforms, and the absence of memory translation mechanisms or deep cache hierarchies further reduces the sources of indeterminism. From another perspective, and as detailed in [BTG+18], commercial MPSoCs exhibit a high degree of heterogeneity regarding their memory subsystem which is comprised of a rich set of different types of memory (e.g. DRAM, SRAM, QSPI Flash, etc.), each accessed through different bus paths and memory controllers, providing varying degrees of latency and bandwidth guarantees. Although this study focused on a high-end Cortex-A platform, this heterogeneity is also true and even more pronounced in modern MCU-based platforms.

Taking these ideas and insights in mind, it is conceivable that it is possible to achieve a high degree of determinism on MCU AMP virtualization, through an informed and thoughtful layout of VM memory. This is accomplished by distributing data and code segments from different VM through different memory elements, each with dedicated controllers accessed via bus paths which enable fully concurrent accesses or assigned in such a way that minimizes contention.

4.6.1 Contention-Aware Memory Layout

The Musca-A chip's memory subsystem and interconnect analysis (see Section 3.1.2.1), supported by a set of empirical observations (see Section 5.6) allows to come up with a memory layout, which is intended to minimize contention of shared resources. Targeting the AMP configuration (Section 4.3.6) and prioritizing the secure VM (RTOS) running on CPU0, the process starts with an idealistic layout scenario and gets iteratively rearranged until all VMs' memory is allocated.

For very low memory footprint systems, all code and data for both VMs would completely fit in iSRAM, resulting in no contention. In this case, iSRAM0 is assigned to secure VM and iSRAM3 to non-secure VM, given their tightly coupled nature, and distributing the remaining SRAM elements according to each VM memory needs, maintaining the invariant that each one is exclusively assigned to a single VM. As a FreeRTOS image compiled with only a small toy application amounts to about 25KB, this would only be feasible when only minimal functionality was included in each VM. In a more realistic setting, there is the need to offload code segments to the external memories. If possible, one of the VMs full images is maintained in the iSRAM and the other's code is migrated to the eSRAM. If not, both migrate. At first sight, assigning one of the VMs to QSPI flash would minimize contention as there would be no sharing of a controller. However, sharing eSRAM results in better performance and less contention, given that QSPI is clocked at a much lower frequency. Observations have shown that when a core is accessing this memory, on a concurrent request, the latter will be stalled for a longer period until the former is served, as the bus expansion port is also shared. The eSRAM's 2MB are believed to suffice for hosting both VMs' code. If not, the non-secure VM shall be placed, or even partially the secure VM, in QSPI. This results in the worst scenario regarding both performance and contention. Note that the impact of moving both code segments to the external memories, despite sharing the same bus expansion master port or the same controller, will not greatly increase contention when good code locality is present, as

instruction caches are assumed to will always be enabled in both cores. This is not guaranteed, however, since a compromised OS running on non-secure VM (CPU1) could disable caches and increase contention on the bus expansion connected to the code memories. Although in the Musca-A platform, the non-secure software cannot access the cache control register, this continues to be true, as it can execute in such a way that continuously thrashes cache lines.

5. Evaluation and Results

The evaluation was conducted on an ARM Musca-A Test Chip Board running both cores at 50 MHz. More details regarding the hardware platform are described in Section 3.1.2. The hypervisor was configured to run two FreeRTOS (version 9.0.0) [Fre] instances as secure and non-secure VMs. In all experiments, with exception to the ones presented in Section 5.6, the lLTZVisor hypervisor is evaluated for a single-core and an AMP configuration. This means that five different test case scenarios were set as listed: (i) native FreeRTOS; FreeRTOS as (ii) secure VM OS instance and (iii) non-secure VM OS instance in a single-core configuration, and; FreeRTOS as (iv) secure VM OS instance and (v) non-secure VM OS instance in an AMP configuration. Moreover, the memory map was allocated so that the code of both VMs are loaded into the external SRAM (eSRAM) and the respective data into different internal SRAMs (iSRAMs). GNU ARM Embedded Toolchain (version 7-2017-q4-major) was used to compile all instances listed before as well as lLTZVisor. The -O2 optimization flag was included for all cases (except in Sections 5.1 and 5.6 where -O0 was also used).

5.1 Memory Footprint

On the first part of the experiments, the focus was on memory footprint. To assess the hypervisor's size, the size tool of the GNU ARM Embedded Toolchain was used. Results were collected for different levels of compilation optimizations (-O0 and -O2), as well as for different lLTZVisor's configurations (single-core and AMP).

Table 5.1 shows the size of the hypervisor for each of the system's configurations. As it can be seen in both configurations, the hypervisor presents a reduced size in the order of a magnitude of a few kB. Comparing the single-core to the AMP approach, the latter presents a smaller size. Such difference occurs because of

Table 5.1: lLTZVisor memory footprint (bytes)

Hypervisor Configuration	*Memory Footprint (bytes)*			
	.text	*.data*	.bss	Total
Single-core (-O0)	2994	136	0	3130
Single-core (-O2)	2630	136	0	2766
AMP (-O0)	2427	128	32	2587
AMP (-O2)	2189	128	32	2349

the AMP approach introducing a slave layer which eliminates code for handling VM switches, resulting in an overall TCB reduction. A particular note goes for compilation optimizations. In fact, they do not sort a huge effect, since a large amount of lLTZVisor's code is written in assembly.

5.2 Microbenchmarks

This section focus on the evaluation of the micro-operations part of the VM switching process performed by the lLTZVisor hypervisor. In order to collect the clock cycles that each operation takes, the Cortex-M SysTick timer, configured to run with the same clock speed as the CPU, was used to measure the difference between its timer's value before and after each micro-operation. As this difference directly corresponds to the amount of clock cycles the operation underwent, both the cycles and timing results can be gathered and used to classify the micro-operations.

In-depth, the operations behind the VM switching process are:

- **SVC handling** - This operation is the first triggered by the idle task of the secure VM's guest OS. It is used to start the VM switching process;
- **Vector Table Switch** - This operation is part of both world switch major operations. It is performed to efficiently swap the secure world's vector tables explained on the mechanism described on Section 4.3.4;
- **S VM Context Store** - The operation that stores the secure VM's context;
- **NS VM Context Load** - The operation that loads the non-secure VM's context;
- **NS VM Context Store** - The operation that stores the non-secure VM's context;
- **S VM Context Load** - The operation that loads the secure VM's context;

- **Fetch S VM Handler** - This operation is performed to obtain the respective handler of the secure VM's interrupt that triggered the switch process to the secure VM, prior to calling it;
- **Scheduler** - The process after restoration of the secure VM. It starts by the verification of real-time tasks to run and posteriorly triggers a VM switch process.

Table 5.2: Hypervisor performance statistics

World Switch	*Operation*	*Performance* $\bar{x}$	*Time* @50MHz
Switch to NS VM	*(1) SVC handling*	23	460ns
	(2) Vector Table Switch	10	200ns
	(3) S VM Context Store	28	560ns
	(4) NS VM Context Load	64	1280ns
Switch to S VM	*(5) Vector Table Switch*	14	280ns
	(6) NS VM Context Store	74	1480ns
	(7) S VM Context Load	29	580ns
	(8) Fetch S VM Handler	329	6580ns
Scheduler	*(9) Asymmetric Policy*	40	800ns
	Total	**611**	**12220ns**

The results can be consulted on Table 5.2, revealing that a complete VM switching process takes around 12.22 microseconds, assuming the RTOS is always idle in this context. Every result obtained was deterministic and presented no deviation. The reason behind this is that the portion of code responsible to handle this process (monitor) is promoted to execute on a TCM bank, which provides the CPU low-latency memory accesses without the use of caches that could bring unpredictability.

5.3 Performance

In this section, the focus is on performance of the VM's guest OSes. To assess the performance overhead introduced by the lLTZVisor, five different test case scenarios were ran using the Thread-Metric Benchmark Suite [Log] described on Section 3.3.1. For each benchmark, the score represents the impact on the guest RTOS of the running VM, where higher scores correspond to a smaller impact. All seven benchmarks were ran into the five different test case scenarios laid out at the beginning of this Chapter. When running the benchmark in one VM, the

opposing VM has no active task (i.e. the workload is null) but the SysTick is kept active. This means any side-effect resulting from partial or total starvation is not taken into consideration in this experiments; starvation is evaluated in Section 5.5). For the first part of the experiments, the SysTick of all VM OS instances was configured with a period of 1ms.

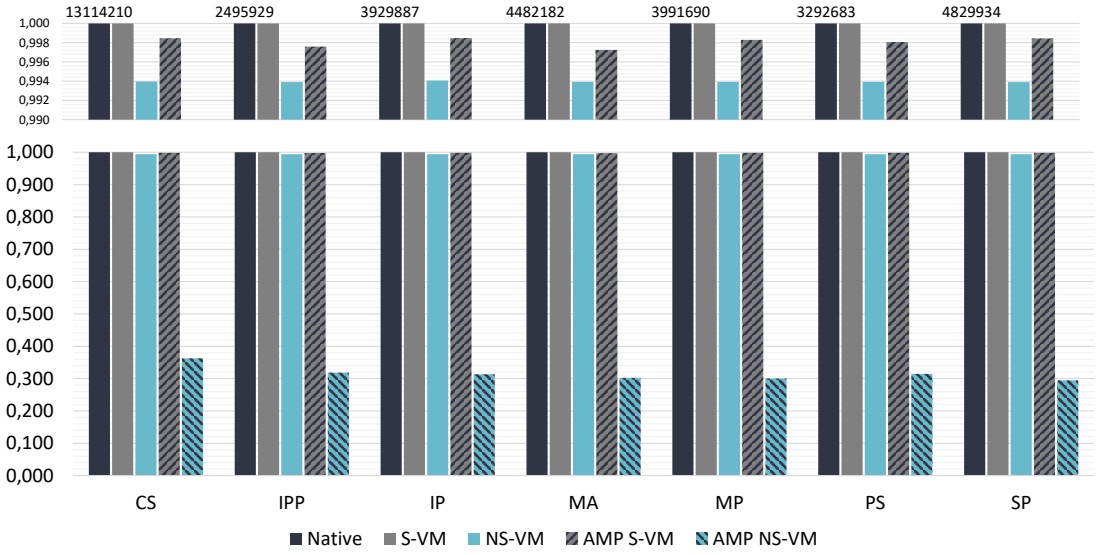


Figure 5.1: Performance for Thread-Metric benchmarks.

Figure 5.1 presents the achieved results, where each bar corresponds to the average relative performance of 10000 collected samples (each sample takes 30 seconds). The values on top of the bars correspond to the average absolute performance. As it can be seen, for the single-core configuration, the secure VM has no performance penalty (asymmetric design principle), while the non-secure VM presents, on average, a performance degradation of about 0.6%. This performance degradation is the result of the periodic preemption imposed by the secure VM. Comparing achieved results with related work for Cortex-A processors (e.g., LTZVisor [PPG+17c]), it is clear the performance overhead decrease (from 2% on LTZVisor to 0.6% on this solution) due to the hardware optimizations of ARM microcontrollers' TrustZone architecture for faster transitioning. Finally, regarding the AMP configuration, there are two notes worth mentioning. First, the AMP secure VM, although running without any hypervisor interference, presents a small performance degradation when compared to native execution, and which varies across the different benchmarks. This degradation is related to contention on shared resources, which will be discussed into detail in Section 5.6. Second, the AMP non-secure VM performance is significantly reduced. This is not related to the virtualization overhead, but mainly due to the execution of the non-secure VM on the secondary core (CPU1) which is inherently slower when running at

the base frequency. A comparison between a native execution of FreeRTOS on CPU0 and CPU1 demonstrated a decrease of performance on the same order of magnitude.

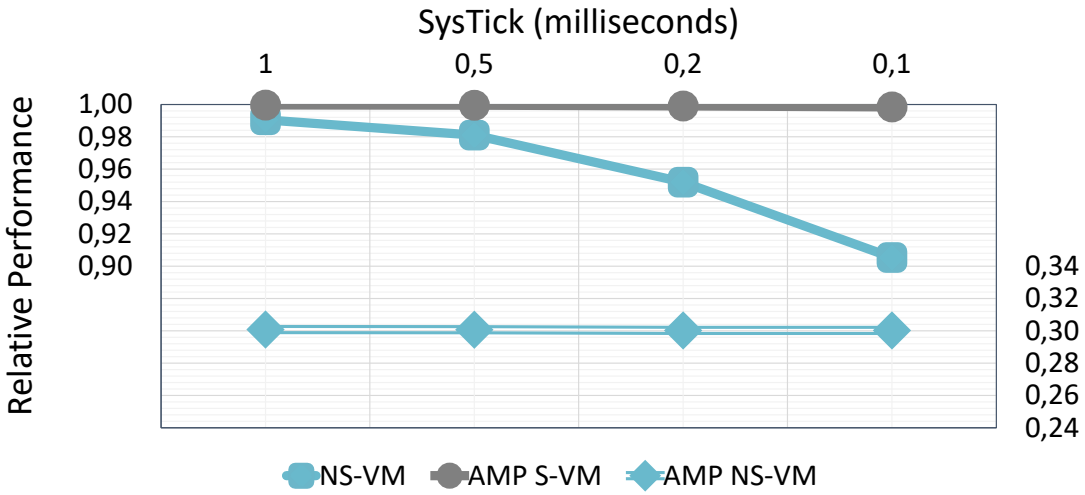


Figure 5.2: Relative performance with different tick rates.

In the second part of the experiments, the focus is on evaluating the dependency between the SysTick rate of one VM and the performance overhead of the opposing VM. The previous experiments were repeated, but instead for three different SysTick rates ranging from 1ms to 100μs. Each point corresponds to the geometric mean of measured results for the seven benchmarks, encompassing a total of 70000 samples per point. From Figure 5.2 it can be concluded that (i) in the single-core configuration the performance of the non-secure VM decreases as the SysTick rate of the secure VM increases while (ii) in the AMP configuration the performance of each VM slightly decreases as the SysTick rate of the other VM increases. Although this phenomenon is not noticeable in Figure 5.2, a performance degradation of about 0.15% was observed. Further analysis must be carried out to fully justify that the decrease of performance is not directly related to SysTick itself, but it is instead, a consequence of concurrency while stressing buses when the FreeRTOS goes through the scheduler. The scheduler needs to go through several critical internal data structures (e.g., task and synchronization control blocks) while following different execution paths (e.g., affecting code locality).

5.4 Interrupt Latency

Interrupt latency, which can be defined as the time from the moment an interrupt is triggered until the moment its handler starts to execute, is a critical metric for

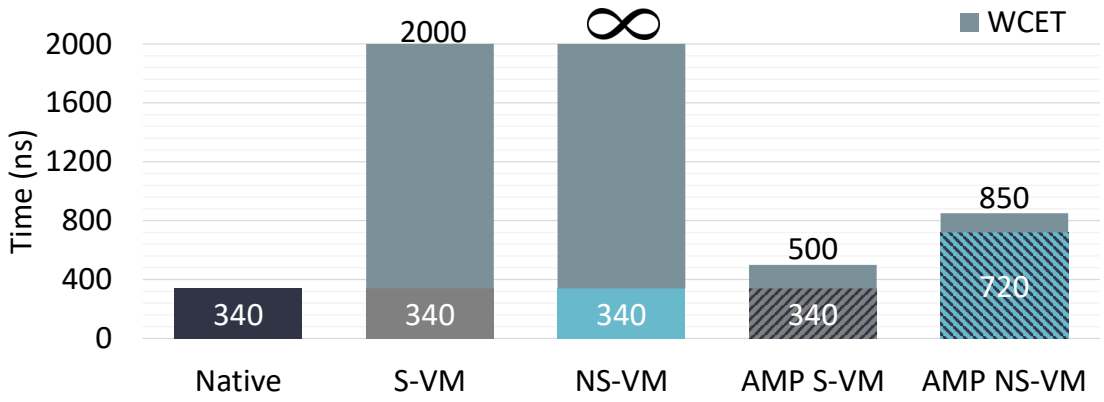


Figure 5.3: Interrupt latency.

real-time systems. To assess the interrupt latency, a dedicated timer was setup to trigger an interrupt every 10 ms, while guaranteeing this is enough time for the interrupt to be serviced and resume previous execution. As the timer is configured in decrementing, auto-reload fashion, latency is obtained directly by reading the counter register at the beginning of the interrupt handler. Latency for the five different test case scenarios was measured. For the single-core configuration, the best and the worst case scenarios were taken into consideration, i.e. when an interrupt is directly handled by the VM, as well as when it is mediated by the lLTZVisor. For example, for the secure VM, measurements were made for the interrupt latency when an interrupt is triggered while the secure VM is running, as well as while the non-secure VM is running (i.e., a world switch needs to be performed). All measurements were repeated 10000 times. Figure 5.3 shows the assessed results, which expresses the best- and the Worst-Case Execution Time (WCET). From the collected data, it is clear the additional overhead introduced in a single-core configuration. This is naturally understandable as both VMs necessarily need to share the same CPU, which requires an additional world switch. Notwithstanding, while for the secure VM the WCET has a deterministic upper bound, for non-secure VM this is not necessarily true. The lemniscate symbol on top of the bar means that the non-securre VM interrupt latency, on the WCET, has no specific upper bound, due to the possible starvation imposed by the secure VM (see Section 5.6). Notwithstanding, this limitation was naturally taken into consideration, which is why this architecture envisions the use of a soft real-time or IoT-enabled OS as a non-secure VM. Regarding the test cases for the AMP configuration, each VM handles directly its own interrupts without any hypervisor interference. However, there are two facts that still deserve an explanation: (i) first, the additional jitter on both cases is mainly explained by concurrency on memory and buses, and; (ii) second, the increased value for the AMP non-secure

VM is related to the fact the VM is executed on the CPU1, a phenomenon already observed in the previous section.

5.5 Starvation

The asymmetric design principal (single-core configuration), borrowed from SafeG [SHT10] and LTZVisor [PPG+17c], ensures the secure VM has a greater scheduling priority than the non-secure VM. While this ensures the timing requirements of the (secure) real-time environment remains nearly intact, it also gives rise to two issues: isolation and starvation. In this subsection the focus is on observing and evaluating starvation. Experiments presented in Section 5.3 were repeated, but using different workloads. A real-time task to the FreeRTOS instance running as secure VM was added as a way of emulating different workloads on the secure VM. Four different workloads were emulated with utilizations of 0, 25, 50 and 75%. FreeRTOS running as secure VM has the SysTick configured to trigger every millisecond. This means the real-time task will be consuming the CPU for 0, 250, 500 and 750 microseconds, respectively. Each point corresponds to the geometric mean of measured results for the seven benchmarks, encompassing a total of 70000 samples per point.

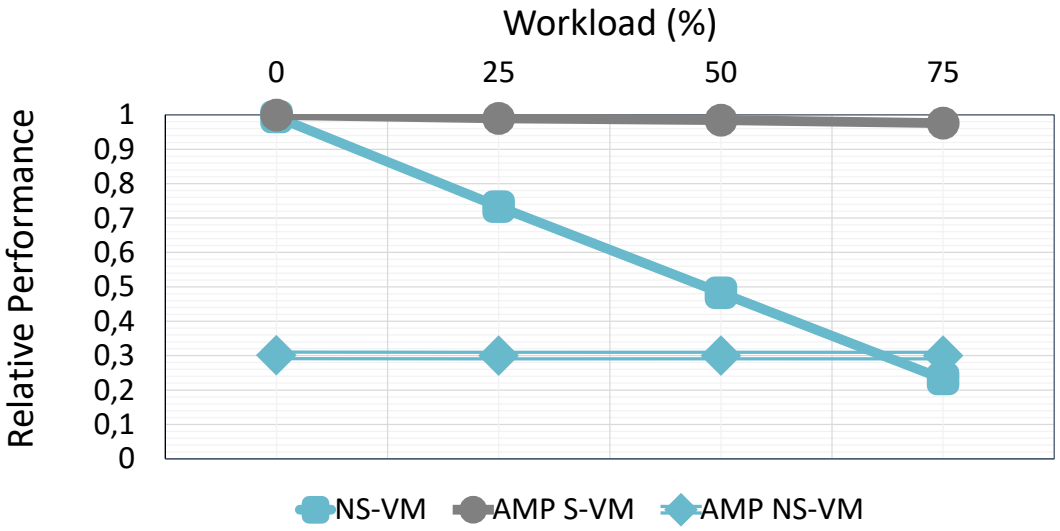


Figure 5.4: Relative performance with different workloads.

From Figure 5.4 it is conclusive that in the single-core configuration, the performance of the non-secure VM decreases linearly as the workload increases. This can lead, in the worst case, to a complete starvation of the non-secure VM, in case the secure VM never releases the CPU. For the AMP configuration, although not

noticeable on the graph, there is a slight performance decrease which is related to contention (see Section 5.6)

5.6 Contention

Finally, in the last part of these experiments, the focus is on evaluating contention. Obviously, the focus was solely on the AMP configuration and consequently on observing how the secure VM timing predictability may be hampered by non-secure VM execution. To evaluate such interference four different test case scenarios were set:

1. **Pessimist Memory Layout (P)** - where the system designer has no concerns regarding the memory map, and the code of both VMs runs from the QSPI, while the data is placed into a single iSRAM bank;
2. **Common Memory Layout (C)** - where the system is configured to stress concurrency, by running code related to the secure VM and non-secure VM from the eSRAM and QSPI, respectively, and data from different iSRAMs;
3. **Realistic Memory Layout (R)** - which was the memory layout used in all aforementioned experiments (both VMs are loaded into the eSRAM and respective data into different iSRAMs), and finally;
4. **Ideal Memory Layout (I)** - where code and data of the secure VM are small enough to fit within a single tightly coupled iSRAM, and the code and data of the non-secure VM are placed on a eSRAM and a tightly coupled iSRAM, respectively.

In all test cases scenarios, the lLTZVisor runs from the same memory as the secure VM. Moreover, on CPU1 (which is running the non-secure VM) the instruction cache is disabled as a way to maximize contention.

Figure 5.5 presents achieved results. Each bar corresponds to the best and WCET, i.e. when the CPU1 (non-secure VM) is enabled and disabled. A number of 10000 samples were collected per test per bar. From Figure 5.5 it is conclusive that for the pessimistic and common memory layout there is significant interference from the non-secure VM onto the secure VM. The non-secure VM is able to add a considerable interference, which results in a lack of determinism and timing predictability. In contrast, when the memory layout is distributed according to the guidelines proposed in Section 4.6.1, the secure VM suffers none or really small

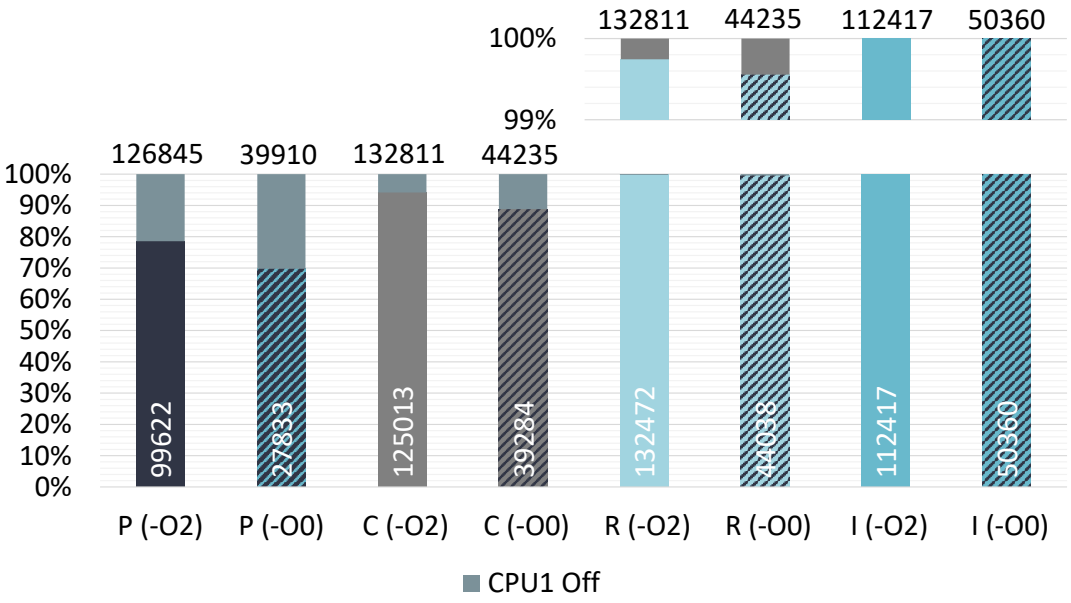


Figure 5.5: Contention.

interference (ideal and realistic, respectively) from the non-secure VM. Additionally, it is seen that the higher are the compilation optimizations, the smaller the contention gets. This is related to the reduction in the number of memory access instructions. Finally, it is worth mentioning that the experiments related to the interrupt latency for the four test case scenarios were repeated and it was observed that the interrupt latency follows a similar pattern of the measured performance. For example, for an ideal memory layout, the jitter on the secure VM interrupt latency is also non-existent.

6. Conclusion

Although low-end and low-cost embedded devices sill present a limited amount of virtualization solutions to this date, it is seen that virtualization techniques have been a game-changer to mid- and high-end embedded applications, addressing multiple situations where different levels of criticality are required and where the dramatical increase of engineering costs due to growing functionalities of ESs are a concern. Several alternatives have been studied to create an environment where multiple environments can be combined into the same platform, from which virtualization technology stands out as the de-facto solution, while benefiting from characteristics such as reliability, safety and security.

Given the high demand of the embedded enterprise world to introduce virtualization to their products, processor manufacturers were forced to start developing hardware virtualization support in order to reduce costs. TrustZone technology, despite not being virtualization-oriented, has been seen as a way to guarantee a strong isolation and consolidation to mixed-criticality systems, as seen on LTZVisor. This technology stands out due to its wide presence in a great part of processors used in the embedded market, that is dominated by ARM-based CPUs. Following this, ARM's low-end range of processors received recently their respective security extensions, the TrustZone for microcontrollers, which sparked early on a particular interest in exploiting its capabilities to create a refactored version of LTZVisor targeting microcontrollers, an approach that up until now is not available on MCUs.

In this thesis lLTZVisor was presented as a viable lightweight virtualization infrastructure for low-end and low-cost systems. The TrustZone technology for microcontrollers was shown how it can be effectively exploited to provide isolation on mixed-criticality systems, which are expected to be deployed on billions of tomorrow's ARM microcontrollers. Furthermore, multi-core technology is addressed by the lLTZVisor in an AMP configuration, demonstrating advantages in terms of

computing power and schemes to minimize contention and improve predictability for modern MCU-based platforms.

The implemented hypervisor was evaluated on a reference low-end ARM multi-core platform, and assessed results demonstrate reduced memory footprint, deterministic and low timing results of the operations behind the VM transitioning process, and little to no performance penalty introduced on the VMs running on top of the lLTZVisor hypervisor. Also, interrupt latencies on the WCET scenario demonstrated a slight introduced overhead due to the share of CPU for the single-core approach and concurrency on memory and buses for the AMP approach. The impact caused by raising the workload of the VMs in the single-core approach was shown to lead in the worst case to a complete starvation of the non-secure VM, and a slight performance decrease for both VMs related to contention in the AMP approach. To conclude, four different test scenarios were set in order to purposefully demonstrate the interference that each memory layout scheme causes on the results of determinism and timing predictability of the lLTZVisor AMP.

6.1 Future Work

Regardless of the developed hypervisor providing the fundamentals for future research on the topic of TrustZone-assisted virtualization for low-end devices, it is believed that there are still improvements left to be made on lLTZVisor. Apart from adding an Inter-VM communication mechanism to the lLTZVisor hypervisor, future work will mainly focus on scalability and power consumption improvements.

Firstly, an alternative of single-core multi-guest support is envisioned to be implemented, and then, focus would be on exploring a hybrid solution between TrustZone and para-virtualization to address scalability in terms of multi-guest and multi-core. Such envisioned architecture would target a complete isolation between the hypervisor and the VMs, relegating the VMs to the non-secure world and then preserve them in the secure world's memory address space when not running. Consequently, this expects to solve the lack of isolation existing between the hypervisor and the secure VM, while at the same time shrinking the TCB of the system size of the hypervisor. Despite this, on a multi-core configuration this would not necessarily translate into addressing scalability in terms of number of guests, whereas only a great collaboration between TrustZone and para-virtualization could bring a real scalability.

Finally, since energy is a key metric for constrained low-end devices, i.e. IoT sensor nodes, an in-depth evaluation of power consumption must be done, onto posteriorly exploring synergy between performance levels and energy efficiency. Where additionally, in such study, the pros and cons would be compared to other hardware-based approaches: MPU-based solutions and hardware virtualization support available on modern ARMv8-R processors.

References

[AH10] Alexandra Aguiar and Fabiano Hessel. Embedded systems' virtualization: The next challenge? In *Proceedings of 2010 21st IEEE International Symposium on Rapid System Protyping*, pages 1–7, June 2010.

[Ami04] Amit Singh. An Introduction to Virtualization. `http://www.kernelthread.com/publications/virtualization/`, 2004. Online; accessed: 1-August-2018.

[ARMa] ARM. Arm MPS2+ FPGA Prototyping Board. `https://developer.arm.com/products/system-design/development-boards/fpga-prototyping-boards/mps2`. Online; accessed: 27-August-2018.

[ARMb] ARM. Arm Musca-A Test Chip Board. `https://developer.arm.com/products/system-design/development-boards/iot-test-chips-and-boards/musca-a-test-chip-board`. Online; accessed: 28-August-2018.

[ARMc] ARM. Software Development Without a Hardware Target. `https://developer.arm.com/products/system-design/fixed-virtual-platforms`. Online; accessed: 27-August-2018.

[ARM09] ARM. Building a Secure System using TrustZone® Technology. *ARM Security Technology*, pages 1–108, 2009.

[ARM15] ARM. Fast Models. *Fixed Virtual Platforms (FVP) Reference Guide*, pages 1–79, 2015.

[ARM16a] ARM. ARMv8-M Exception Handling. *ARM Technical Paper*, pages 1–28, 2016.

[ARM16b] ARM. Introduction to the ARMv8-M Architecture. *ARM Technical Paper*, pages 1–26, 2016.

[ARM16c] ARM. TrustZone technology for ARMv8-M Architecture. *ARM Technical Paper*, pages 1–37, 2016.

[ARM17] ARM. ARM Cortex-M33 Processor. *Technical Reference Manual*, pages 1–158, 2017.

[ARM18a] ARM. Arm CPU Architecture. `https://developer.arm.com/products/architecture/cpu-architecture`, 2018. Online; accessed: 14-August-2018.

[ARM18b] ARM. ARM Musca-A Test Chip and Board. *Technical Reference Manual*, pages 1–117, 2018.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[BTG+18] Ayoosh Bansal, Rohan Tabish, Giovani Gracioli, Renato Mancuso, Rodolfo Pellizzoni, and Marco Caccamo. Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, page 55, 2018.

[CBS+17] Alfons Crespo, Patricia Balbastre, José Simó, Javier Coronel, Daniel Gracia Perez, and Philippe Bonnot. Hypervisor Feedback Control of Mixed Critical Systems: the XtratuM Approach. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2017.

[DGM09] David Déharbe, Stephenson Galvão, and Anamaria Martins Moreira. Formalizing FreeRTOS: First Steps. In Marcel Vinícius Medeiros Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications*, pages 101–117, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, November 2004.

[Exp] Express Logic. ThreadX RTOS Real-Time Operating System. `https://rtos.com/solutions/threadx/real-time-operating-system/`. Online; accessed 28-August-2018.

[FA16] Muhammad Farooq-i-Azam and Muhammad Naeem Ayyaz. Embedded Systems Security. *CoRR*, abs/1610.00632, 2016.

[FLWH10] Torsten Frenzel, Adam Lackorzynski, Alexander Warg, and Hermann Härtig. ARM TrustZone as a Virtualization Technique in Embedded Systems. *Twelfth Real-Time Linux Workshop*, 2010.

[FMR11] Konstantinos Fysarakis, Harry Manifavas, and Konstantinos Rantos. Embedded Systems Security, Aspects of secure embedded systems design and implementation. September 2011.

[Fre] FreeRTOS. About the FreeRTOS Kernel. `https://www.freertos.org/RTOS.html`. Online; accessed: 28-August-2018.

[Gen] General Dynamics. Hypervisor Products. `https://gdmissionsystems.com/products/secure-mobile/hypervisor`. Online; accessed 28-August-2018.

[GYCH18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, April 2018.

[Hei07] Gernot Heiser. Virtualization for Embedded Systems. *Open Kernel Labs Technology White Paper*, 2007.

[Hei08] Gernot Heiser. The Role of Virtualization in Embedded Systems. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, pages 11–16, New York, NY, USA, April 2008. ACM.

[Hei11] Gernot Heiser. Virtualizing embedded systems - why bother? In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 901–905, June 2011.

[HHF+05] Hermann Hartig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza secure-system architecture. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2005.

[HL10] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. *Proceedings of the first ACM asia-pacific workshop ...*, pages 19–23, August 2010.

[HSH+08] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park,

Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *2008 5th IEEE Consumer Communications and Networking Conference*, pages 257–261, January 2008.

[IBM07] IBM. Virtualization in Education. *IBM White Paper*, pages 1–20, 2007.

[KLJ+13] Se Won Kim, Chiyoung Lee, MooWoong Jeon, Hae Young Kwon, Hyun Woo Lee, and Chuck Yoo. Secure device access for automotive software. In *International Conference on Connected Vehicles and Expo (ICCVE)*, 2013.

[Koo04] Philip Koopman. Embedded System Security. *Computer*, 37(7):95–97, July 2004.

[KR17] Hyoseung Kim and Ragunathan Rajkumar. Predictable Shared Cache Management for Multi-Core Real-Time Virtualization. *ACM Transactions on Embedded Computing Systems*, 17(1):22:1–22:27, December 2017.

[KYK+08] Wataru Kanda, Yu Yumura, Yuki Kinebuchi, Kazuo Makijima, and Tatsuo Nakajima. SPUMONE: Lightweight CPU Virtualization Layer for Embedded Systems. In *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, volume 1, pages 144–151, December 2008.

[Log] Express Logic. Measuring Real-Time Performance Of An RTOS. *Thread-Metric Benchmark Suite*, pages 1–20.

[MAC+17] José Martins, João Alves, Jorge Cabral, Adriano Tavares, and Sandro Pinto. uRTZVisor: A Secure and Safe Real-Time Hypervisor. *Electronics*, 6(4), 2017.

[MBBP18] Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms. In *IEEE Int. Conf. on Industrial Technology*, pages 1651–1657, February 2018.

[MSY16] Tim Menasveta, Diya Soubra, and Joseph Yiu. Introducing ARM Cortex-M23 and Cortex-M33 Processors with TrustZone for ARMv8-M. *Cortex-M Product Marketing*, pages 1–9, 2016.

[NMB+16] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. TrustZone Explained: Architectural Features and Use

Cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, November 2016.

[OS15] Gabriel Cephas Obasuyi and Arif Sari. Security Challenges of Virtualization Hypervisors in Virtualized Hardware Environment. *International Journal of Communications, Network and System Sciences*, Vol.08No.07, 2015.

[PG74] Gerald Popek and Robert Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[PGP+17] Sandro Pinto, Tiago Gomes, Jorge Pereira, Jorge Cabral, and Adriano Tavares. IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices. *IEEE Internet Computing*, 21(1):40–47, January 2017.

[POP+14] Sandro Pinto, Daniel Oliveira, Jorge Pereira, Nuno Cardoso, Mongkol Ekpanyapong, Jorge Cabral, and Adriano Tavares. Towards a lightweight embedded virtualization architecture exploiting ARM TrustZone. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–4, September 2014.

[PPG+17a] Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong, and Adriano Tavares. Towards a TrustZone-Assisted Hypervisor for Real-Time Embedded Systems. *IEEE Computer Architecture Letters*, 16(2):158–161, July 2017.

[PPG+17b] Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong, and Adriano Tavares. Towards a TrustZone-Assisted Hypervisor for Real-Time Embedded Systems. *IEEE Comp. Arch. Letters*, 16(2):158–161, 2017.

[PPG+17c] Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. LTZVisor: TrustZone is the Key. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:22, Dagstuhl, Germany, June 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[PPRP18] Runyu Pan, Gregor Peach, Yuxin Ren, and Gabriel Parmer. Predictable Virtualization on Memory Protection Unit-Based Microcontrollers. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 62–74, April 2018.

[PS18] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys*, preprint, 2018.

[PW08] Sri Parameswaran and Tilman Wolf. Embedded systems security—an overview. *Design Automation for Embedded Systems*, 12(3):173–183, September 2008.

[Rus08] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. 42:95–103, January 2008.

[Sch14] Mathijs Jeroen Scheepers. Virtualization and Containerization of Application Infrastructure : A Comparison. 2014.

[SHT10] Daniel Sangorrin, Shinya Honda, and Hiroaki Takada. Dual operating system architecture for real-time embedded systems. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Brussels, Belgium*, pages 6–15, 2010.

[SR04] John Stankovic and Ragunathan Rajkumar. Real-Time Operating Systems. *Real-Time Syst.*, 28(2-3):237–253, November 2004.

[THAC18] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems. In *Proceedings of the 55th Annual Design Automation Conference (DAC)*, pages 98:1–98:6, June 2018.

[VMW06] VMWare. Virtualization Overview. *VMWare White Paper*, pages 1–11, 2006.

[VMW07] VMWare. A Performance Comparison of Hypervisors. *VMWare Technical Paper*, pages 1–22, 2007.

[Web] Webtechmag. Free Virtualization Software & Hypervisors. `http://webtechmag.com/wp-content/uploads/2017/08/virtual-architecture.jpg`. Online; accessed on: 01-August-2018.

[WJM08] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin. Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, October 2008.

[Xen18] Xen. Xen ARM with Virtualization Extensions. *Xen White Paper*, 2018. Online; accessed: 12-August-2018.

[XTP+17] Meng Xu, Linh Thi, Xuan Phan, Hyon-Young Choi, and Insup Lee. vCAT: Dynamic Cache Management Using CAT Virtualization. In

IEEE Real-Time and Embedded Technology and Applications Symposium, pages 211–222, 2017.

[Yiu15] Joseph Yiu. ARMv8-M Architecture Technical Overview. *ARM White Paper*, pages 1–16, 2015.

[Yiu16] Joseph Yiu. The Next Steps in the Evolution of Embedded Processors for the Smart Connected Era. *Embedded World 2016*, pages 1–10, 2016.